Programming Emerging Storage Interfaces

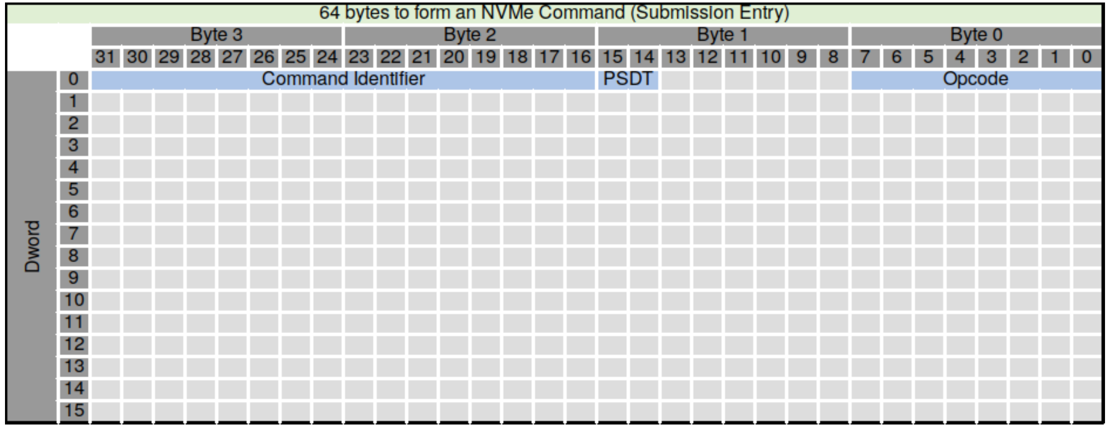
VAULT 2020 | Simon A. F. Lund | Samsung | SSDR

<simon.lund@samsung.com>

Programming Emerging Storage Interfaces: Why?

- The device media changed
- The device interface changed
  - Command Response Protocol
  - Queues
    - Submission Entries
    - Completions Entries

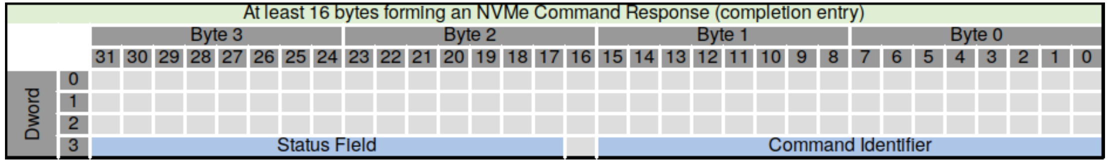
Command: 64byte Submission Queue Entry (**sqe**)



64 bytes to form an NVMe Command (Submission Entry)

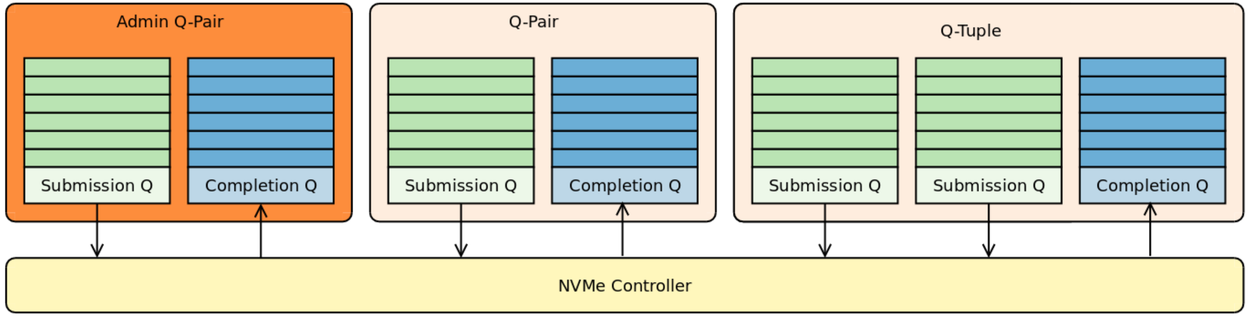
Dword	Byte 3	Byte 2	Byte 1	Byte 0
	31 30 29 28 27 26 25 24	23 22 21 20 19 18 17 16	15 14 13 12 11 10 9 8	7 6 5 4 3 2 1 0
0	Command Identifier	Command Identifier	PSDT	Opcode
1				
2				
3				
4				
5				
6				
7				
8				
9				
10				
11				
12				
13				
14				
15				

Response: (at least) 16byte Completion Queue Entry (**cqe**)



At least 16 bytes forming an NVMe Command Response (completion entry)

Dword	Byte 3	Byte 2	Byte 1	Byte 0
	31 30 29 28 27 26 25 24	23 22 21 20 19 18 17 16	15 14 13 12 11 10 9 8	7 6 5 4 3 2 1 0
0				
1				
2				
3	Status Field	Status Field	Command Identifier	Command Identifier



Programming Emerging Storage Interfaces: Why?

- **New** devices doing **old** things *faster*
  - The software storage-stack becomes the bottleneck
  - Requires: **efficiency**

Programming Emerging Storage Interfaces: Why?

- **New** devices doing **old** things *faster*
  - The software storage-stack becomes the bottleneck
  - Requires: **efficiency**
- **New** devices doing **old** things in a **new** way
  - Responsibilities trickle up the stack
  - Host-awareness, the higher up, the higher the benefits
  - Device ➔ OS Kernel ➔ Application
  - Requires: **control**, as in, commands other than read/write

Programming Emerging Storage Interfaces: Why?

- **New** devices doing **old** things *faster*
  - The software storage-stack becomes the bottleneck
  - Requires: **efficiency**
- **New** devices doing **old** things in a **new** way
  - Responsibilities trickle up the stack
  - Host-awareness, the higher up, the higher the benefits
  - Device ➔ OS Kernel ➔ Application
  - Requires: **control**, as in, commands other than read/write
- **New** devices doing **new** things!
  - New storage semantics such as Key-Value
  - New hybrid semantics introducing compute on and near storage
  - Requires: **flexibility / adaptability**, as in, ability to add new commands

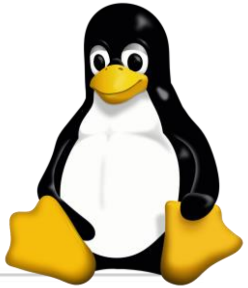
Programming Emerging Storage Interfaces: Why?

- **New** devices doing **old** things *faster*
  - The software storage-stack becomes the bottleneck
  - Requires: **efficiency**
- **New** devices doing **old** things in a **new** way
  - Responsibilities trickle up the stack
  - Host-awareness, the higher up, the higher the benefits
  - Device ➔ OS Kernel ➔ Application
  - Requires: **control**, as in, commands other than read/write
- **New** devices doing **new** things!
  - New storage semantics such as Key-Value
  - New hybrid semantics introducing compute on and near storage
  - Requires: **flexibility / adaptability**, as in, ability to add new commands

➔Increased requirements on the host software stack

Programming Emerging Storage Interfaces: Using io_uring

- **The newest Linux IO interface: io_uring**
  - A user space ⇔ kernel communication channel
  - A transport mechanism for commands



Programming Emerging Storage Interfaces: Using io_uring

- **The newest Linux IO interface: io_uring**
  - A user space ⬄ kernel communication channel
  - A transport mechanism for commands

- Queue Based (ring mem. kernel ⬄ user space)
  - Submission queue

    - populated by user space, consumed by Kernel
  - Completion queue

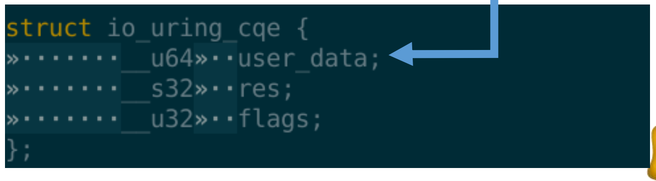
    - populated by kernel, in-response

    - consumed by user space

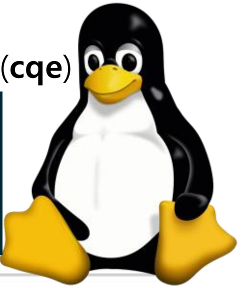
Command: 64byte Submission Queue Entry (**sqe**)

```
struct io_uring_sqe {
»·······uint8_t»opcode;
»·······uint8_t»flags;
»·······uint16_t»·······ioprio;
»·······int32_t»fd;
»·······union {
»·······»·······uint64_t»·······off;
»·······»·······uint64_t»·······addr2;
»·······};
»·······uint64_t»·······addr;
»·······uint32_t»·······len;
»·······uint32_t»·······_flags;
»·······uint64_t»·······user_data;

»·······union {
»·······»·······struct {
»·······»·······»·······uint16_t»·······buf_index;
»·······»·······»·······uint16_t»·······personality;
»·······»·······};
»·······»·······uint64_t»·······__pad2[3];
»·······};
};

struct io_uring_cqe {
»·······uint64_t»·······user_data;
»·······int32_t»res;
»·······uint32_t»·······flags;
};
```

Response: 16byte Completion Queue Entry (**cqe**)



```
struct io_uring_cqe {
»·······__u64»··user_data;
»·······__s32»··res;
»·······__u32»··flags;
};
```



Programming Emerging Storage Interfaces: Using io_uring

- **The newest Linux IO interface: io_uring**
  - A user space ⬄ kernel communication channel
  - A transport mechanism for commands
- Queue Based (ring mem. kernel ⬄ user space)
  - Submission queue

    - populated by user space, consumed by Kernel
  - Completion queue

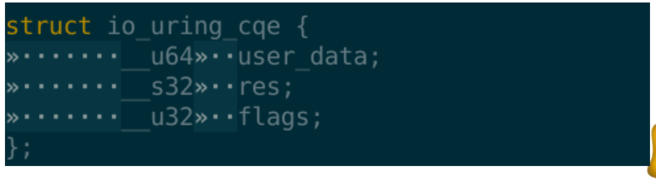
    - populated by kernel, in-response

    - consumed by user space
- A syscall, `io_uring_enter`, for sub.+compl.
- A second for queue setup (`io_uring_setup`)
- Resource registration (`io_uring_register`)

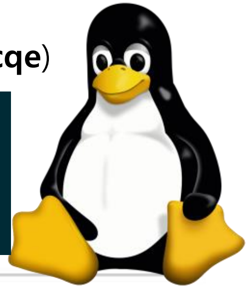
Command: 64byte Submission Queue Entry (**sqe**)

```
struct io_uring_sqe {
»·······uint8_t»opcode;
»·······uint8_t»flags;
»·······uint16_t»·······ioprio;
»·······int32_t»fd;
»·······union {
»·······»·······uint64_t»·······off;
»·······»·······uint64_t»·······addr2;
»·······};
»·······uint64_t»·······addr;
»·······uint32_t»·······len;
»·······uint32_t»·······_flags;
»·······uint64_t»·······user_data;

»·······union {
»·······»·······struct {
»·······»·······»·······uint16_t»·······buf_index;
»·······»·······»·······uint16_t»·······personality;
»·······»·······};
»·······»·······uint64_t»·······__pad2[3];
»·······};
};

struct io_uring_cqe {
»·······uint64_t»·······user_data;
»·······int32_t»res;
»·······uint32_t»·······flags;
};
```

Response: 16byte Completion Queue Entry (**cqe**)



```
struct io_uring_cqe {
»·······__u64»··user_data;
»·······__s32»··res;
»·······__u32»··flags;
};
```



Programming Emerging Storage Interfaces: Using io_uring

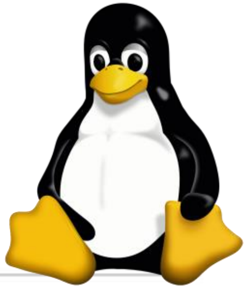
- It is **efficient*** on a single core one can get
  - 1.7M IOPS (polling) ~ 1.2M IOPS (interrupt driven)
  - The Linux aio interface was at ~ 608K IOPS (interrupt driven)

- It is quite **flexible**
  - Works with UNIX file abstraction

    - Not just when it encapsulates block devices
  - Growing command-set (opcodes)

- It is **adaptable**

    - Add a new opcode ➔ implement handling of it in the **Kernel**

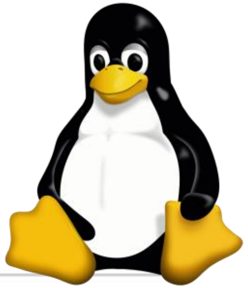


***Efficient IO with io_uring, https://kernel.dk/io_uring.pdf**

Kernel Recipes 2019 - Faster IO through io_uring, https://www.youtube.com/watch?v=-5T4Cjw46ys

Programming Emerging Storage Interfaces: Using io_uring

- Advanced Features
  - Register files (**RF**)
  - Fixed buffers (**FB**)
  - Polling IO (**IOP**)
  - SQ polling by Kernel Thread (**SQT**)

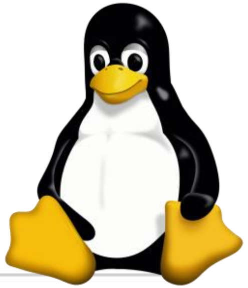


Programming Emerging Storage Interfaces: Using io_uring

- Advanced Features
  - Register files (**RF**)
  - Fixed buffers (**FB**)
  - Polling IO (**IOP**)
  - SQ polling by Kernel Thread (**SQT**)
- Efficiency revisited
  - Null Block instance w/o block-layer

4K Random Read (Interrupt)	Latency (nsec)	IOPS QD1	IOPS QD16
aio	1200	741 K	749 K
io_uring	926	922 K	927 K
io_uring +RF +FB	807	1.05 M	1.02 M

4K Random Read (SQT Polling)	Latency (nsec)	IOPS QD1	IOPS QD16
io_uring +SQT +RF	644	1.25 M	1.7 M
io_uring +SQT RF +FB	567	1.37 M	2.0 M

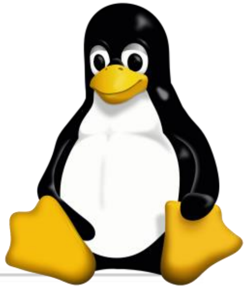


Programming Emerging Storage Interfaces: Using io_uring

- Advanced Features
  - Register files (**RF**)
  - Fixed buffers (**FB**)
  - Polling IO (**IOP**)
  - SQ polling by Kernel Thread (**SQT**)
- Efficiency revisited
  - Null Block instance w/o block-layer
- Efficiency vs Ease of Use
  - Opcode restrictions when using **FB**
  - Do **not** use **IOP** + **SQT**
  - Know that register files is required for **SQT**
  - Use buffer and file registration indexes instead of *iov and handles

4K Random Read (Interrupt)	Latency (nsec)	IOPS QD1	IOPS QD16
aio	1200	741 K	749 K
io_uring	926	922 K	927 K
io_uring +RF +FB	807	1.05 M	1.02 M

4K Random Read (SQT Polling)	Latency (nsec)	IOPS QD1	IOPS QD16
io_uring +SQT +RF	644	1.25 M	1.7 M
io_uring +SQT RF +FB	567	1.37 M	2.0 M



Programming Emerging Storage Interfaces: Using io_uring

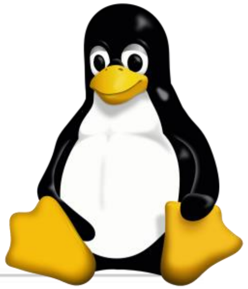
- Advanced Features
  - Register files (**RF**)
  - Fixed buffers (**FB**)
  - Polling IO (**IOP**)
  - SQ polling by Kernel Thread (**SQT**)
- Efficiency revisited
  - Null Block instance w/o block-layer
- Efficiency vs Ease of Use
  - Opcode restrictions when using **FB**
  - Do **not** use **IOP** + **SQT**
  - Know that register files is required for **SQT**
  - Use buffer and file registration indexes instead of *iov and handles

➔**rtfm**, man pages, pdf, mailing-lists, github, and talks document it well

➔**liburing** makes it, if not easy, then *easier*

4K Random Read (Interrupt)	Latency (nsec)	IOPS QD1	IOPS QD16
aio	1200	741 K	749 K
io_uring	926	922 K	927 K
io_uring +RF +FB	807	1.05 M	1.02 M

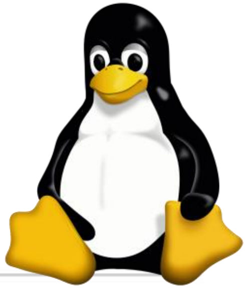
4K Random Read (SQT Polling)	Latency (nsec)	IOPS QD1	IOPS QD16
io_uring +SQT +RF	644	1.25 M	1.7 M
io_uring +SQT RF +FB	567	1.37 M	2.0 M



Programming Emerging Storage Interfaces: Using Linux IOCTLs

- **The oldest? Linux IO interface: IOCTL**
  - A kernel ⇔ user space communication channel

- The interface is
  - **Not** efficient
  - Adaptable but **not** flexible
    - Never break user space!
  - **Control** oriented



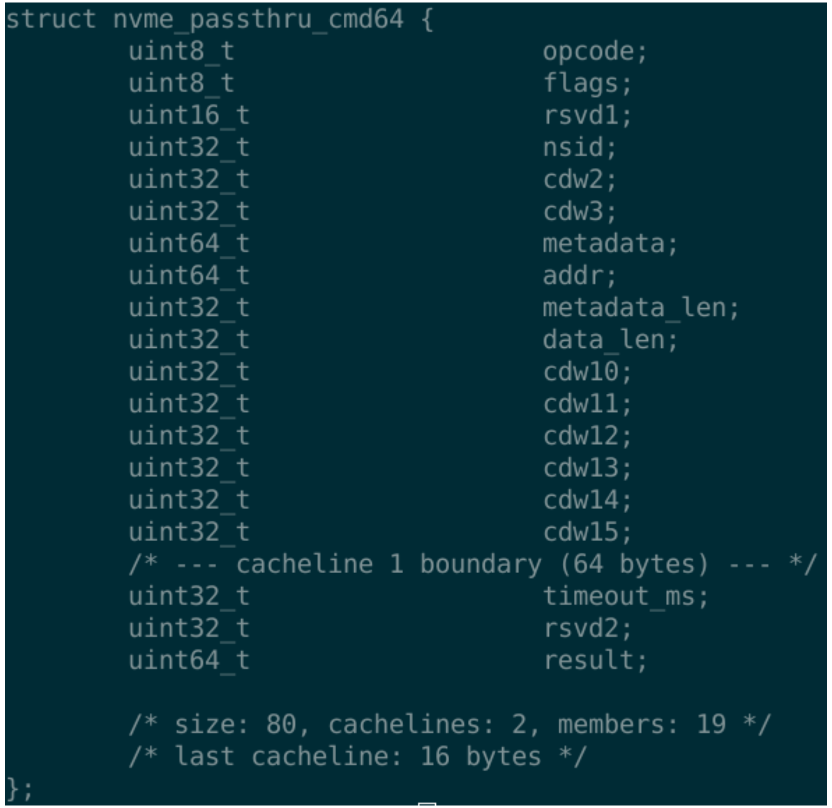
Programming Emerging Storage Interfaces: Using Linux IOCTLs

- **The oldest? Linux IO interface: IOCTL**
  - A kernel ⬄ user space communication channel

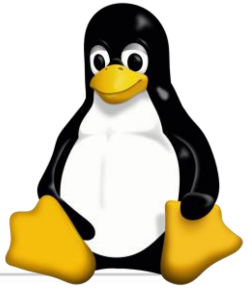
- The interface is
  - **Not** efficient
  - Adaptable but **not** flexible
    - Never break user space!
  - **Control** oriented

- However, the NVMe driver IOCTLs are
  - A transport mechanism for commands
  - Very **flexible** – pass commands without changing the Kernel
  - Rich **control**, but not *full* control, of the NVMe command / **sqe**
  - Can even be used for non-admin IO, however, **not** efficiently

Command: 80byte Submission + Completion



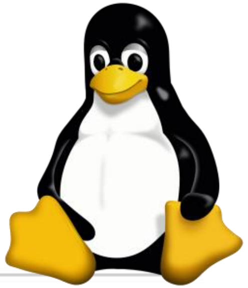
```
struct nvme_passthru_cmd64 {
        uint8_t                         opcode;
        uint8_t                         flags;
        uint16_t                        rsvd1;
        uint32_t                        nsid;
        uint32_t                        cdw2;
        uint32_t                        cdw3;
        uint64_t                        metadata;
        uint64_t                        addr;
        uint32_t                        metadata_len;
        uint32_t                        data_len;
        uint32_t                        cdw10;
        uint32_t                        cdw11;
        uint32_t                        cdw12;
        uint32_t                        cdw13;
        uint32_t                        cdw14;
        uint32_t                        cdw15;
        /* --- cacheline 1 boundary (64 bytes) --- */
        uint32_t                        timeout_ms;
        uint32_t                        rsvd2;
        uint64_t                        result;

        /* size: 80, cachelines: 2, members: 19 */
        /* last cacheline: 16 bytes */
};
```



Programming Emerging Storage Interfaces: Assisted by Linux sysfs

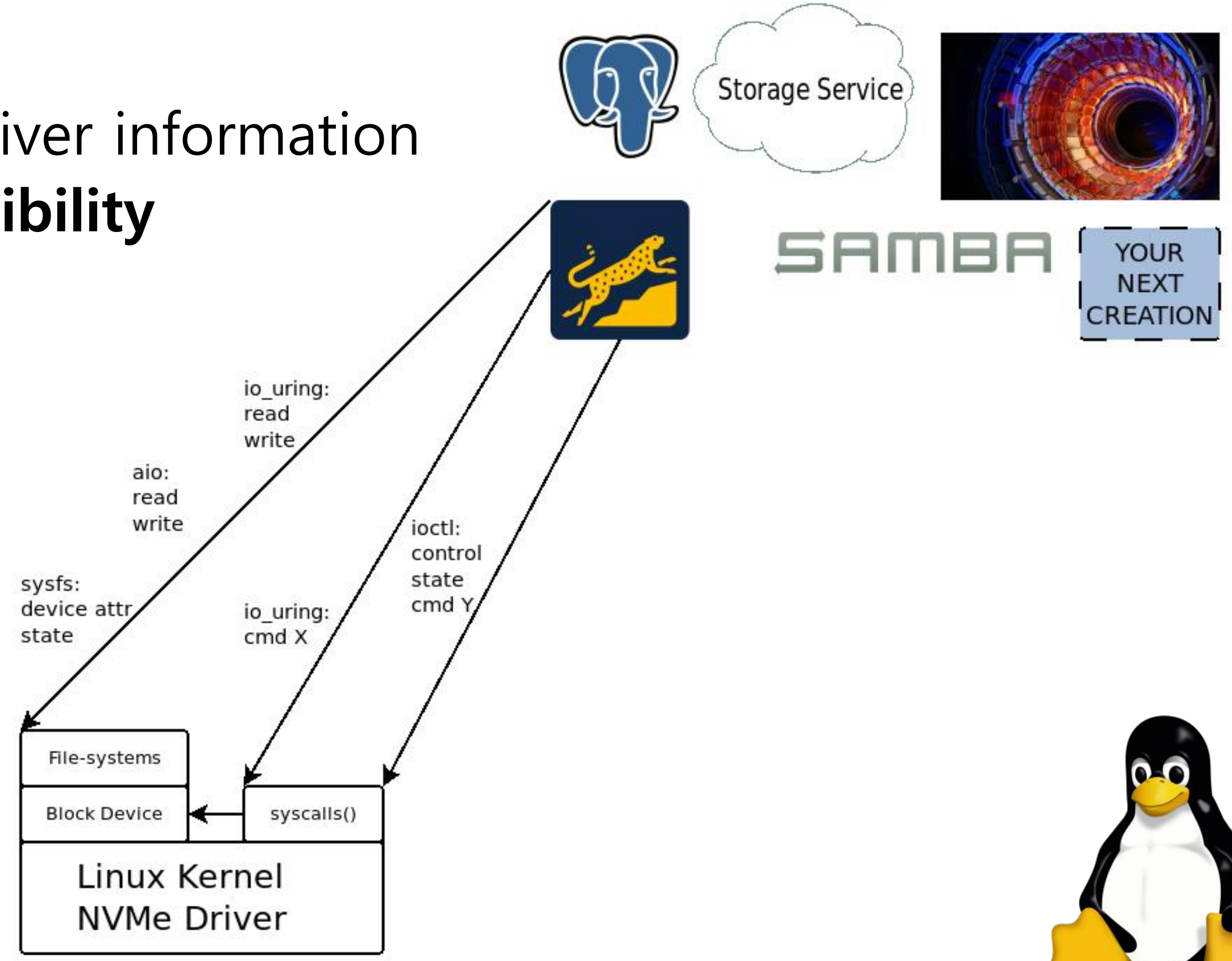
- **The convenient Linux IO interface: sysfs**
  - A kernel ⬄ user space communication channel
  - File system semantics to retrieve system, device, and driver information
    - Great for retrieving device properties

```
root@bullseye:/sys/block/nvme0n1# ls
alignment_offset   hidden     ro
bdi                holders    size
capability         inflight   slaves
dev                integrity  stat
device             mq         subsystem
discard_alignment  nsid       trace
events             power      uevent
events_async       queue      wwid
events_poll_msecs  range
ext_range          removable
root@bullseye:/sys/block/nvme0n1# cat size
28131328
root@bullseye:/sys/block/nvme0n1# 
```



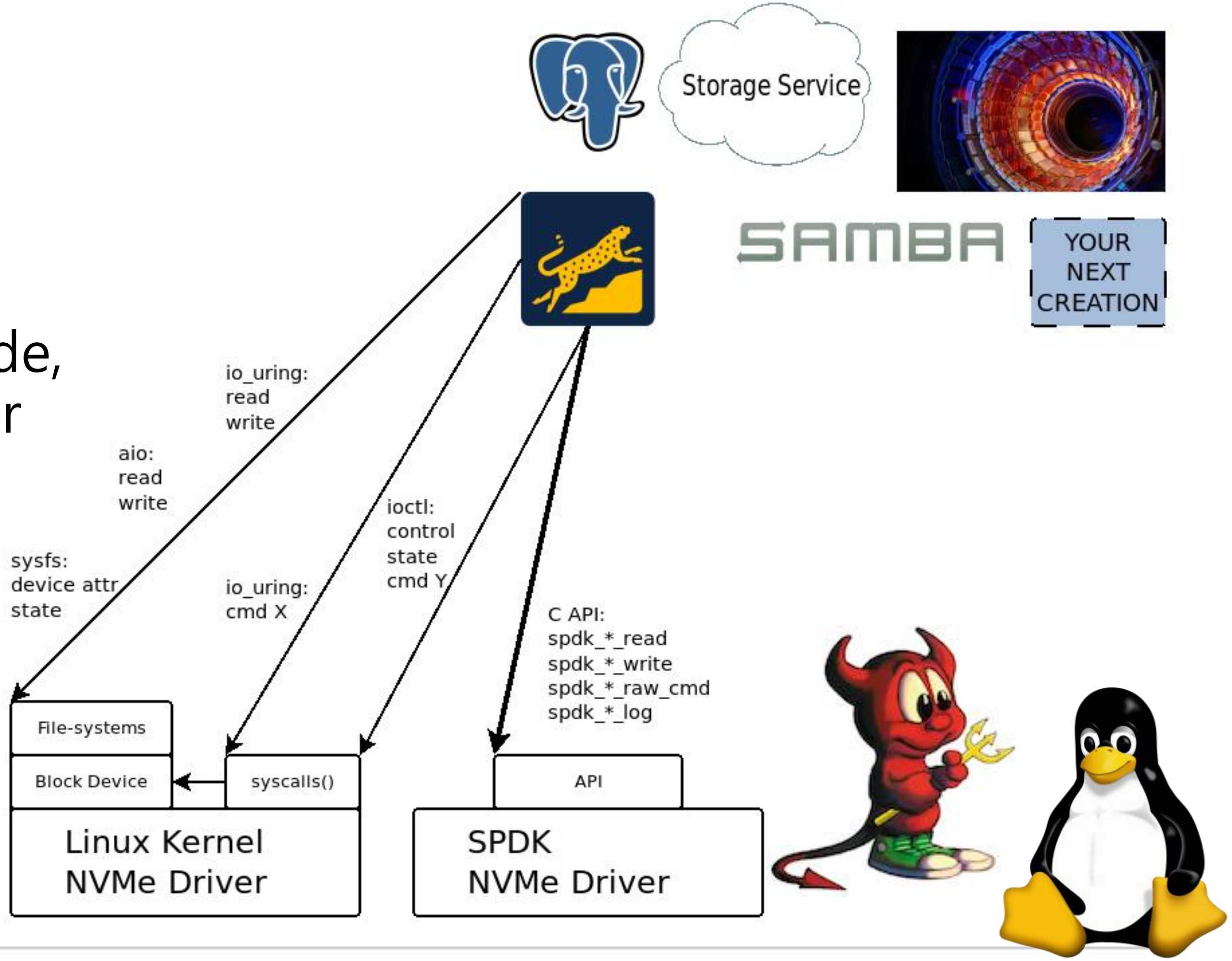
Programming Emerging Storage Interfaces: On Linux

- **Everything you need encapsulated in the file abstraction**
  - io_uring / liburing for **efficiency**
  - **sysfs** for convenient device and driver information
  - NVMe IOCTLs for **control** and **flexibility**



Programming Emerging Storage Interfaces using Intel SPDK

- **The Storage Platform Development Kit**
  - Tools and libraries for high performance, scalable, user-mode storage applications
- It is **efficient***
  - 10M IOPS from one thread
  - Thanks to a user space, polled-mode, asynchronous, lockless NVMe driver
  - **zero-copy** command payloads
- It is **flexible**
  - Storage stack as an API
  - It is extremely **adaptable**
    - Full **control** over SQE construction



Storage Service

SAMBA

YOUR NEXT CREATION

io_uring:
read
write

aio:
read
write

sysfs:
device attr
state

io_uring:
cmd X

ioctl:
control
state
cmd Y

C API:
spdk_*_read
spdk_*_write
spdk_*_raw_cmd
spdk_*_log

File-systems

Block Device

syscalls()

API

Linux Kernel
NVMe Driver

SPDK
NVMe Driver

---

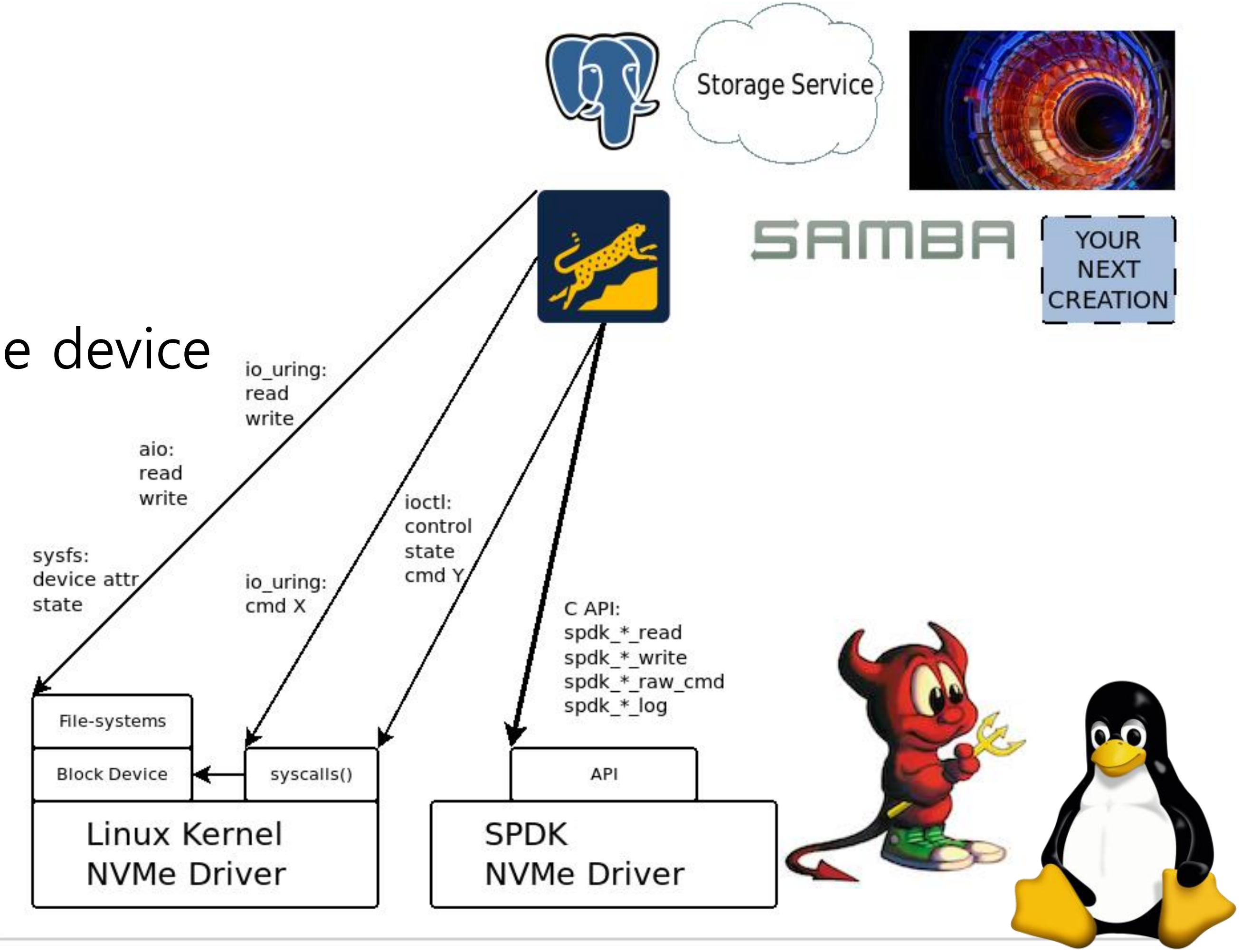
*10.39M Storage I/O Per Second From One Thread, https://spdk.io/news/2019/05/06/nvme/

Programming Emerging Storage Interfaces using Intel SPDK

- **The Storage Platform Development Kit**
  - Tools and libraries for high performance, scalable, user-mode storage applications

- It is **efficient*** revisited
  - 4K Random Read at QD1
  - On **physical** locally attached NVMe device

QD1: io_uring vs SPDK	IOPS	BW
`io_uring +SQT +RF`	**117** K	**479** MB/s
`SPDK`	**150** K	**587** MB/s

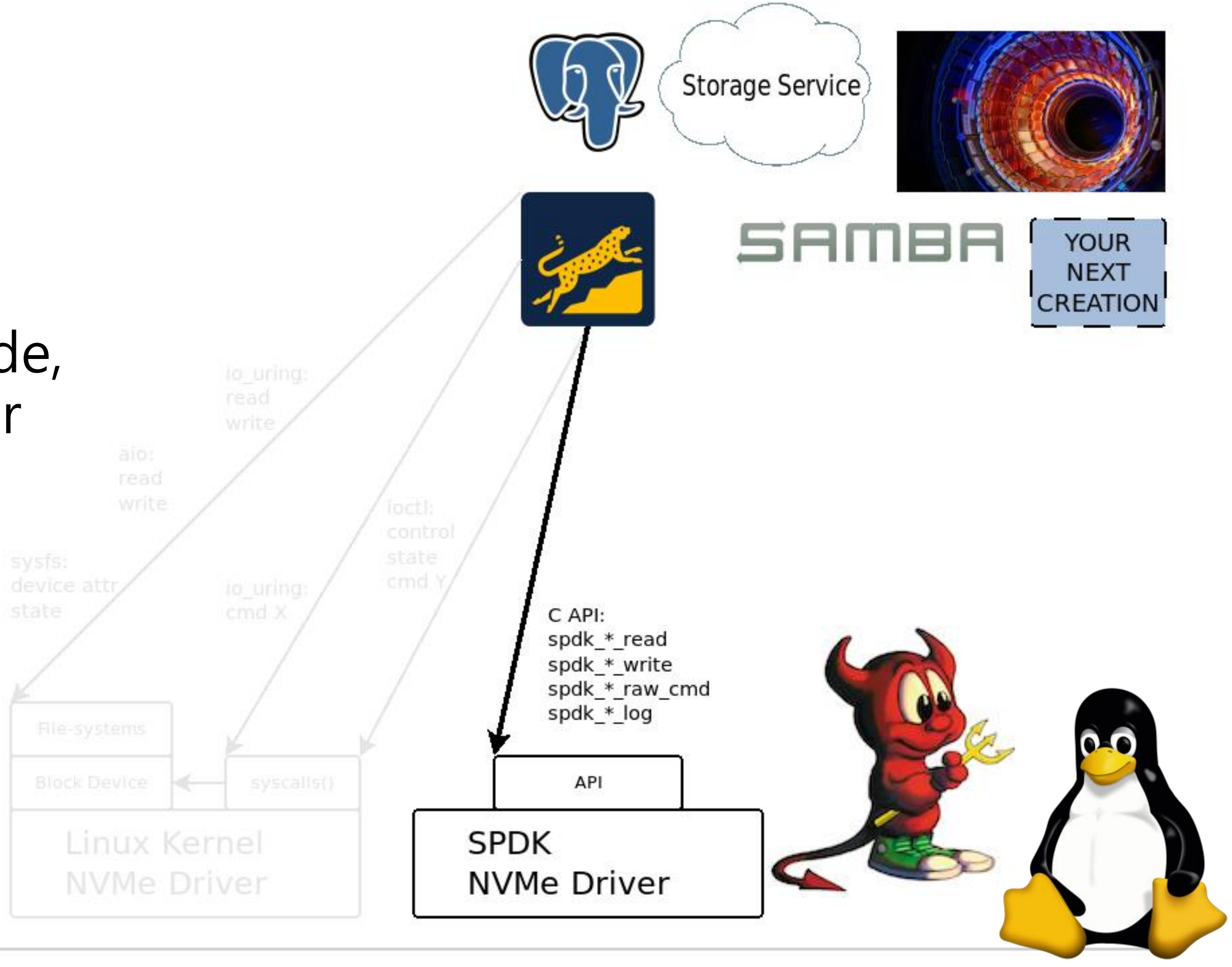


---

*10.39M Storage I/O Per Second From One Thread, https://spdk.io/news/2019/05/06/nvme/

Programming Emerging Storage Interfaces using Intel SPDK

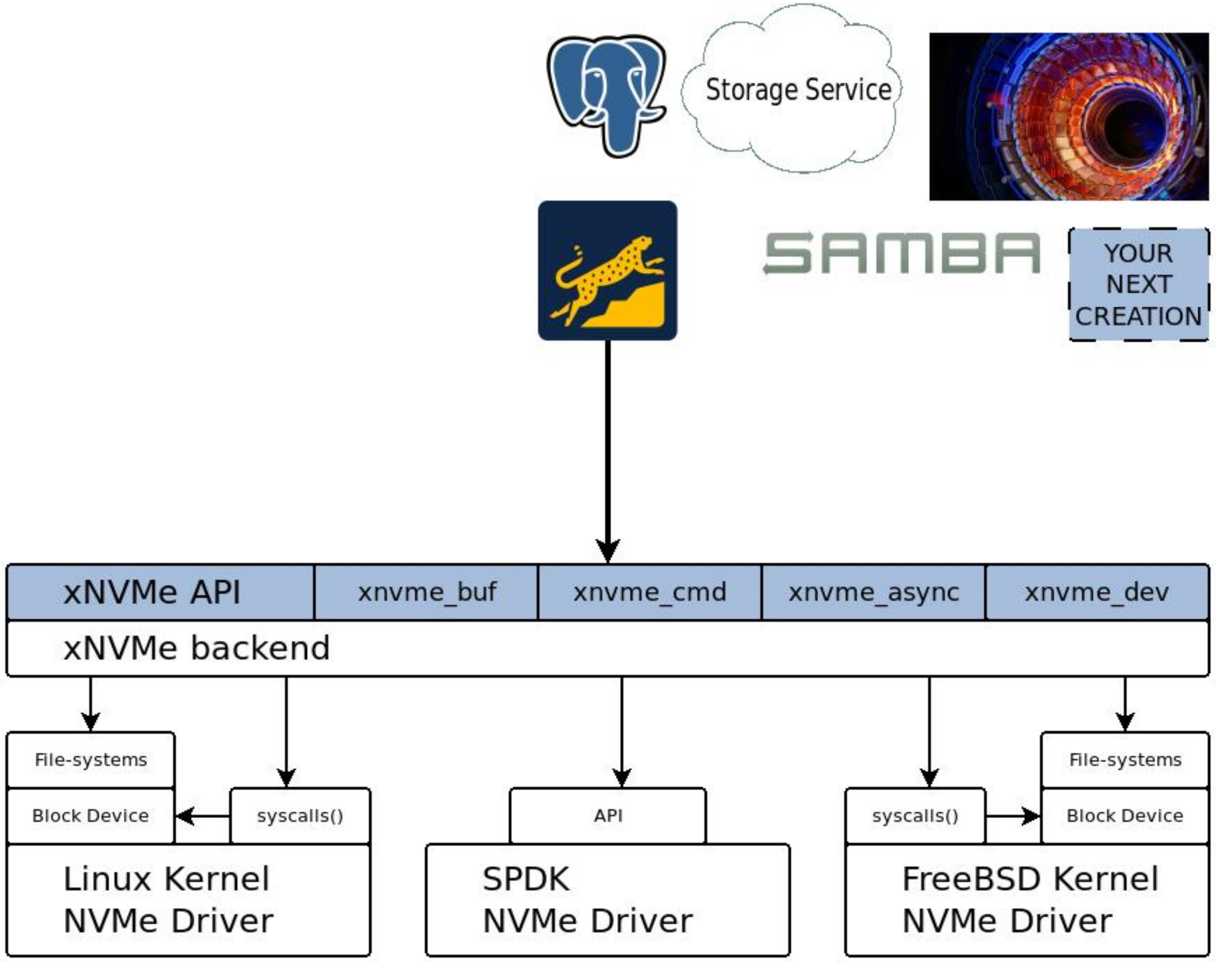
- **The Storage Platform Development Kit**
  - Tools and libraries for high performance, scalable, user-mode storage applications
- It is **efficient***
  - 10M IOPS from one thread
  - Thanks to a user space, polled-mode, asynchronous, lockless NVMe driver
  - **zero-copy** command payloads
- It is **flexible**
  - Storage stack as an API
  - It is extremely **adaptable**
    - Full **control** over SQE construction



---

*10.39M Storage I/O Per Second From One Thread, https://spdk.io/news/2019/05/06/nvme/

Programming Emerging Storage Interfaces using xNVMe



Storage Service

SAMBA

YOUR NEXT CREATION

xNVMe API | xnvme_buf | xnvme_cmd | xnvme_async | xnvme_dev

xNVMe backend

File-systems

Block Device

syscalls()

Linux Kernel
NVMe Driver

API

SPDK
NVMe Driver

syscalls()

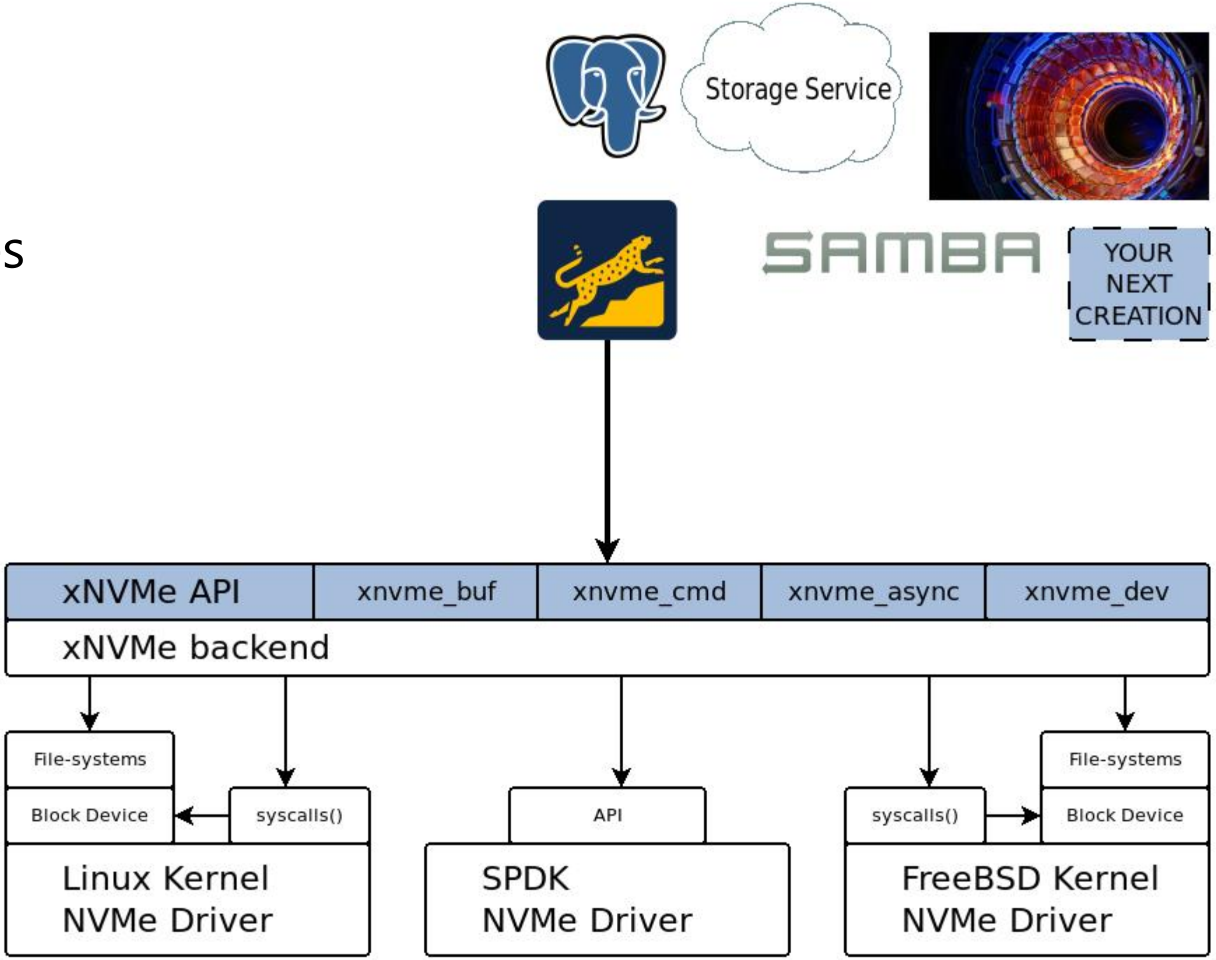
File-systems

Block Device

FreeBSD Kernel
NVMe Driver

Programming Emerging Storage Interfaces using xNVMe

- A unified API **primarily** for **NVMe** devices



Storage Service

SAMBA

YOUR NEXT CREATION

xNVMe API | xnvme_buf | xnvme_cmd | xnvme_async | xnvme_dev

xNVMe backend

File-systems

Block Device

syscalls()

Linux Kernel NVMe Driver

API

SPDK NVMe Driver

syscalls()

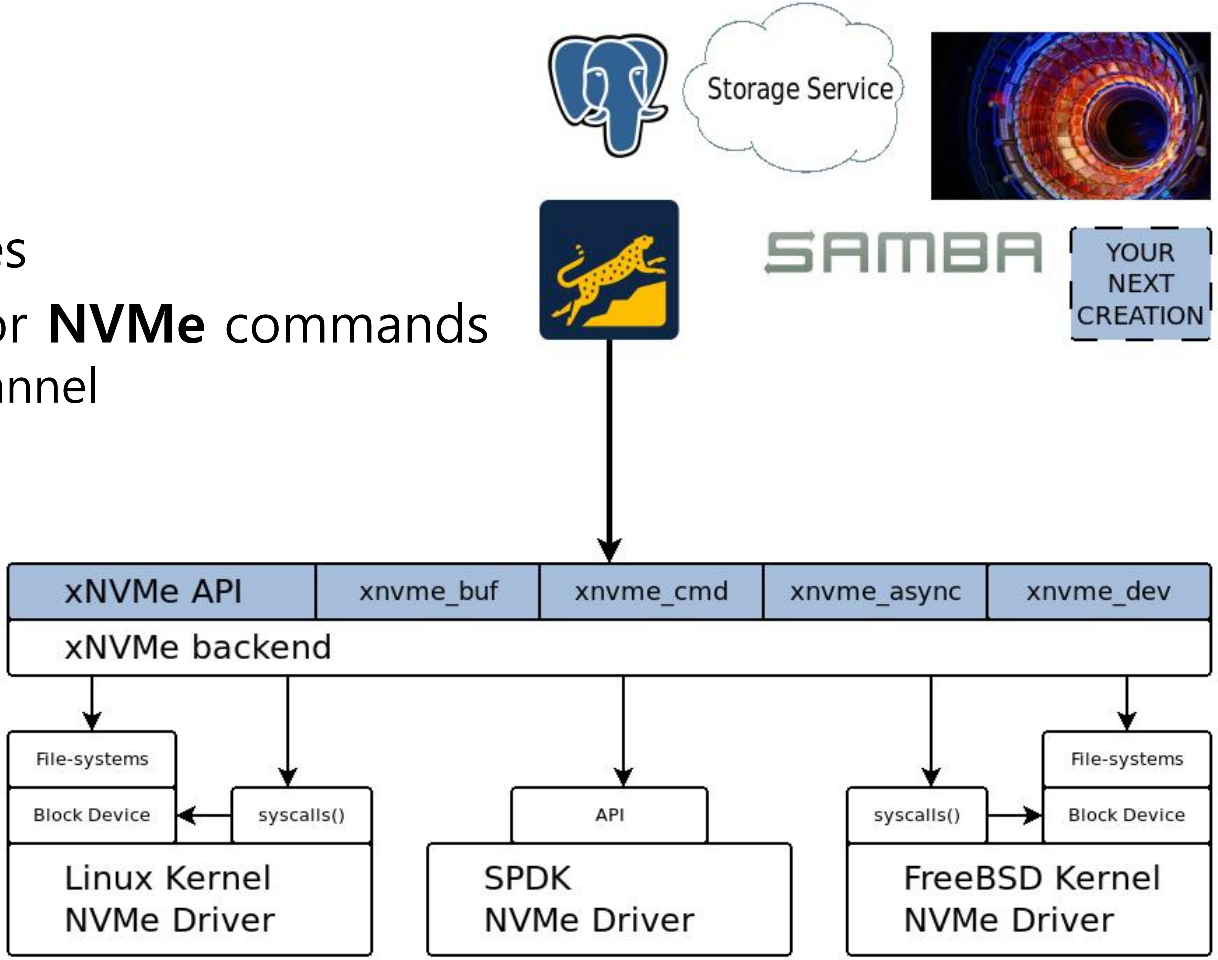
File-systems

Block Device

FreeBSD Kernel NVMe Driver

Programming Emerging Storage Interfaces using xNVMe

- A unified API **primarily** for **NVMe** devices
- A cross-platform transport mechanism for **NVMe** commands
  - A user space ⇔ device communication channel



Storage Service

SAMBA

YOUR NEXT CREATION

xNVMe API	xnvme_buf	xnvme_cmd	xnvme_async	xnvme_dev

xNVMe backend

File-systems

Block Device

syscalls()

Linux Kernel
NVMe Driver

API

SPDK
NVMe Driver

syscalls()

File-systems

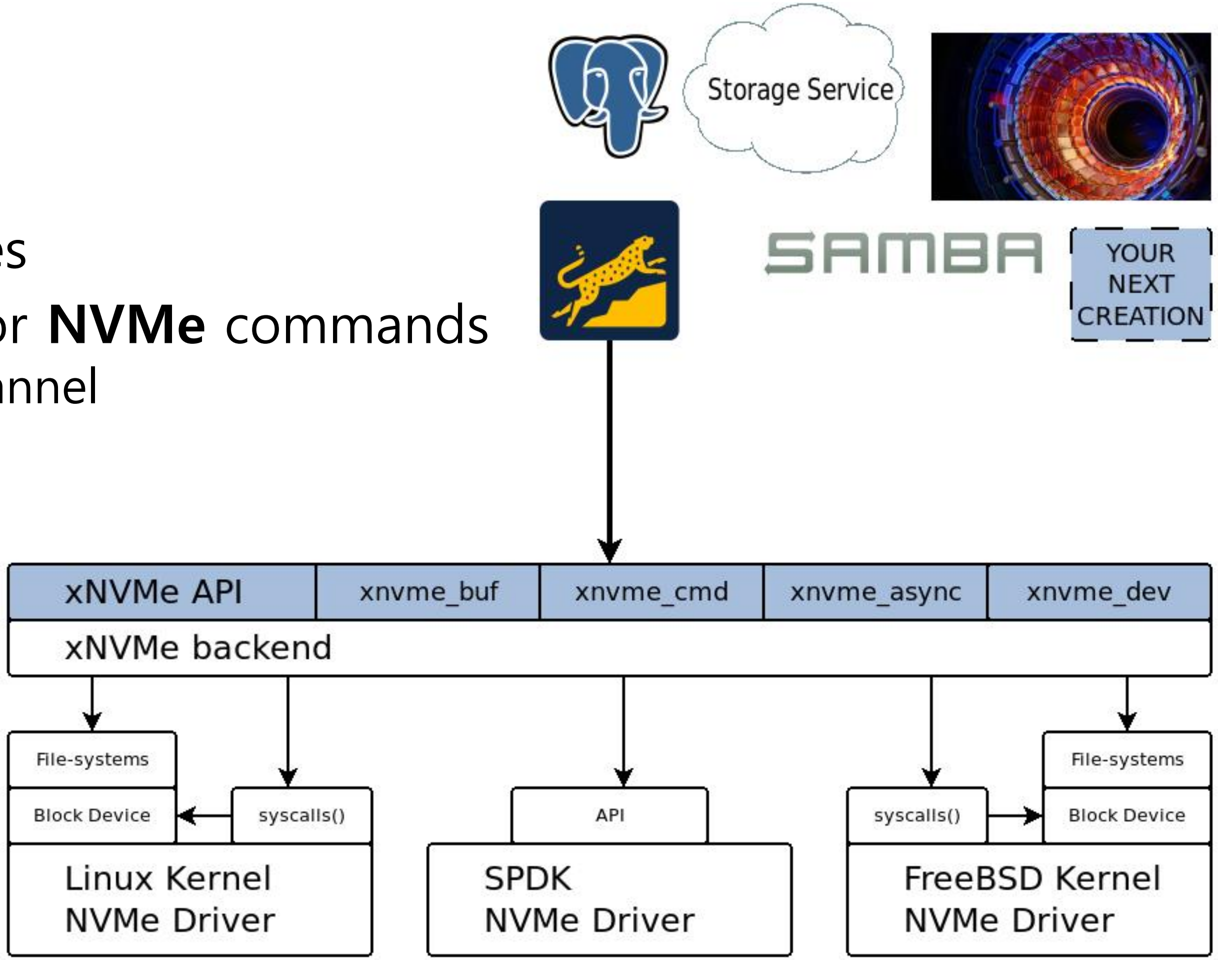
Block Device

FreeBSD Kernel
NVMe Driver

Programming Emerging Storage Interfaces using xNVMe

- A unified API **primarily** for **NVMe** devices
- A cross-platform transport mechanism for **NVMe** commands
  - A user space ⇔ device communication channel

- Focus on being easy to use
  - Reaping the benefits of the lower layers
  - **Without** sacrificing efficiency!
  - ➔ High performance **and** high productivity



Storage Service

SAMBA

YOUR NEXT CREATION

xNVMe API | xnvme_buf | xnvme_cmd | xnvme_async | xnvme_dev

xNVMe backend

File-systems

Block Device

syscalls()

Linux Kernel NVMe Driver

API

SPDK NVMe Driver

syscalls()

File-systems

Block Device

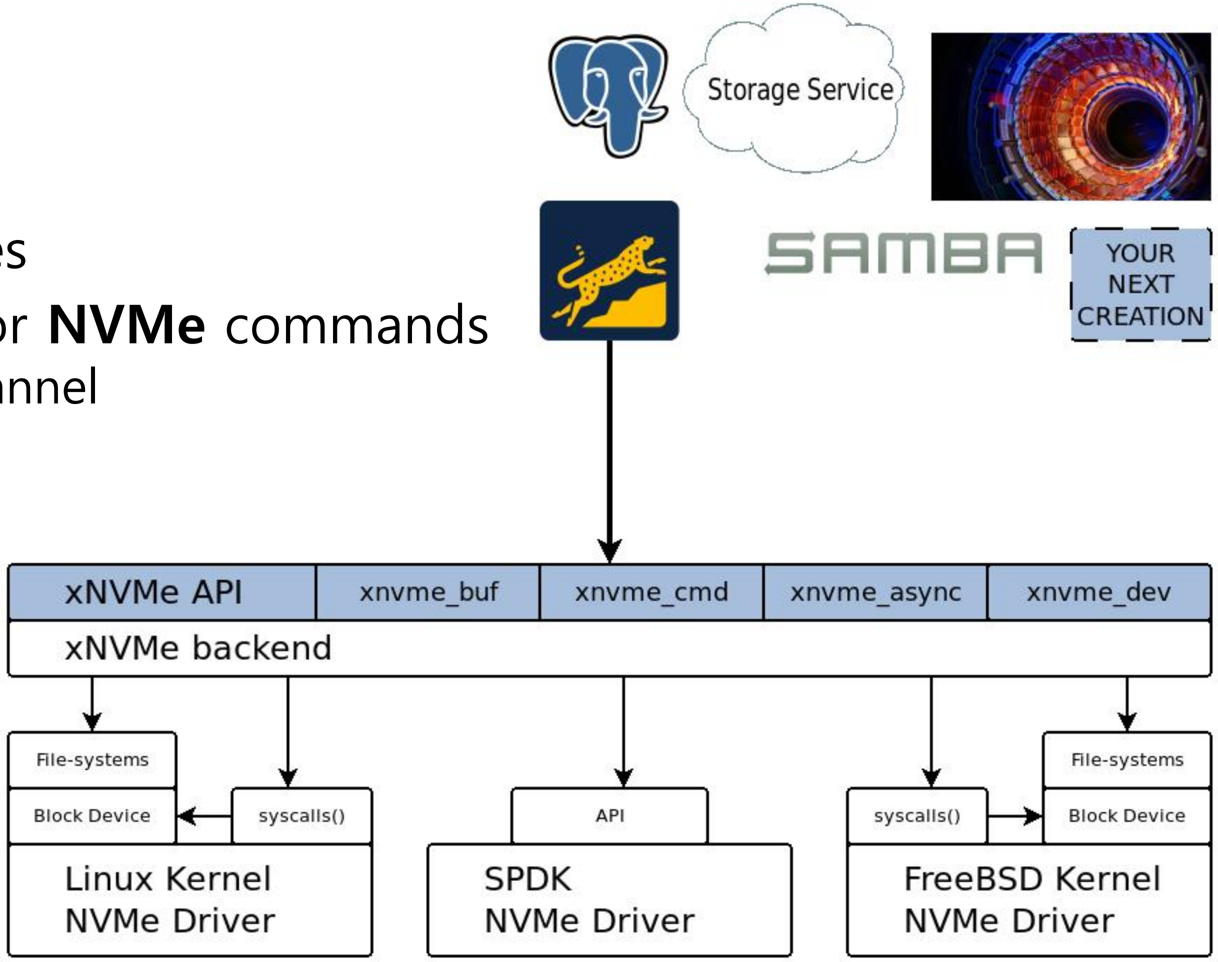
FreeBSD Kernel NVMe Driver

Programming Emerging Storage Interfaces using xNVMe

- A unified API **primarily** for **NVMe** devices
- A cross-platform transport mechanism for **NVMe** commands
  - A user space ⇔ device communication channel

- Focus on being easy to use
  - Reaping the benefits of the lower layers
  - **Without** sacrificing efficiency!
  - ➔ High performance **and** high productivity

- Tools and utilites
  - Including tools to build tools



xNVMe API | xnvme_buf | xnvme_cmd | xnvme_async | xnvme_dev

xNVMe backend

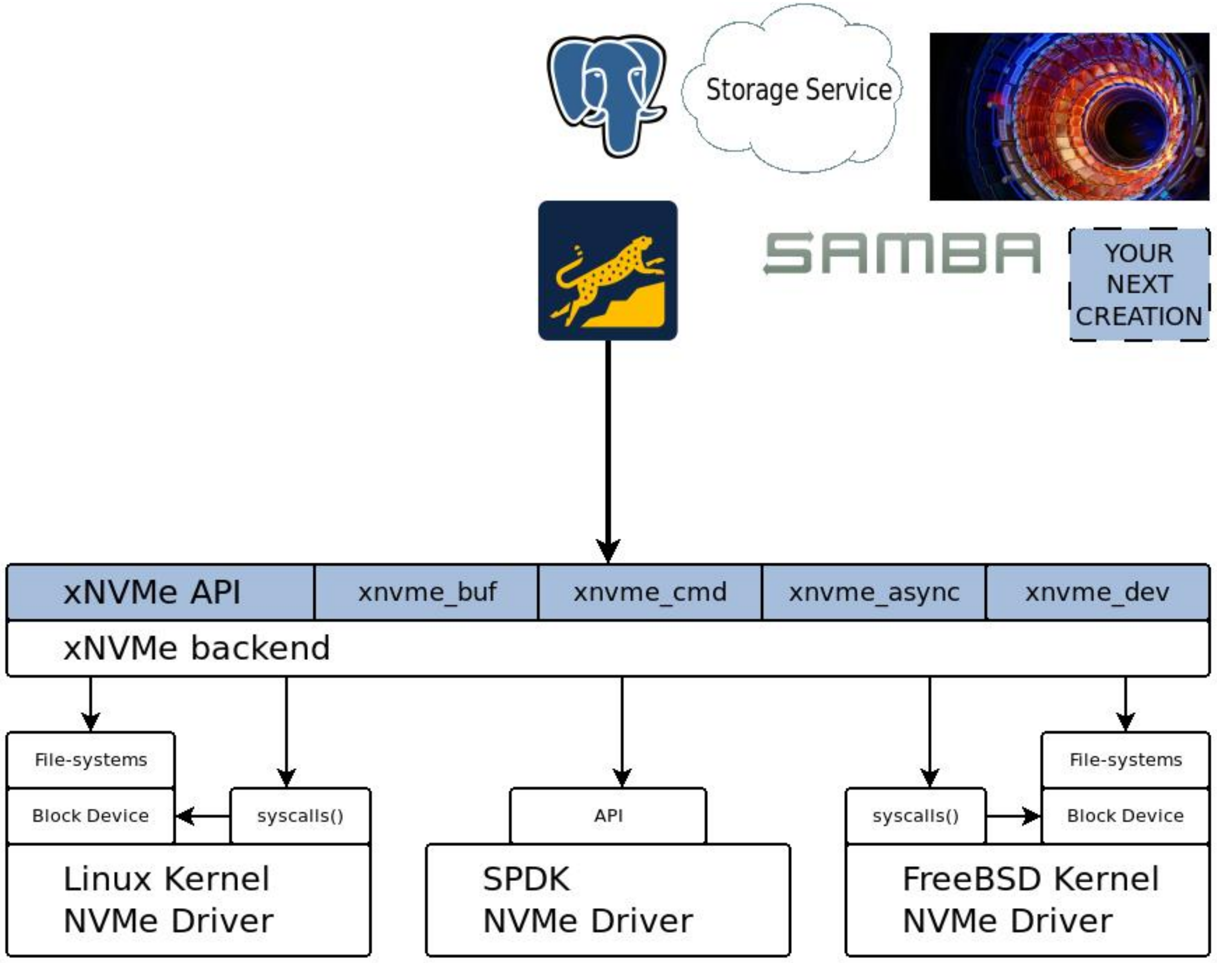
File-systems
Block Device
syscalls()
Linux Kernel NVMe Driver

API
SPDK NVMe Driver

syscalls()
File-systems
Block Device
FreeBSD Kernel NVMe Driver

Programming Emerging Storage Interfaces using the xNVMe API

- **xNVMe Base API**
  - Lowest level interface
- Device
  - Handles
  - Identifiers
  - Enumeration
  - Geometry
- Memory Management
  - Command payloads
  - Virtual memory
- Command Interface
  - Synchronous
  - Asynchronous
    - Requests and callbacks


Storage Service

SAMBA

YOUR NEXT CREATION

xNVMe API | xnvme_buf | xnvme_cmd | xnvme_async | xnvme_dev

xNVMe backend

File-systems

Block Device

syscalls()

Linux Kernel NVMe Driver

API

SPDK NVMe Driver

syscalls()

File-systems

Block Device

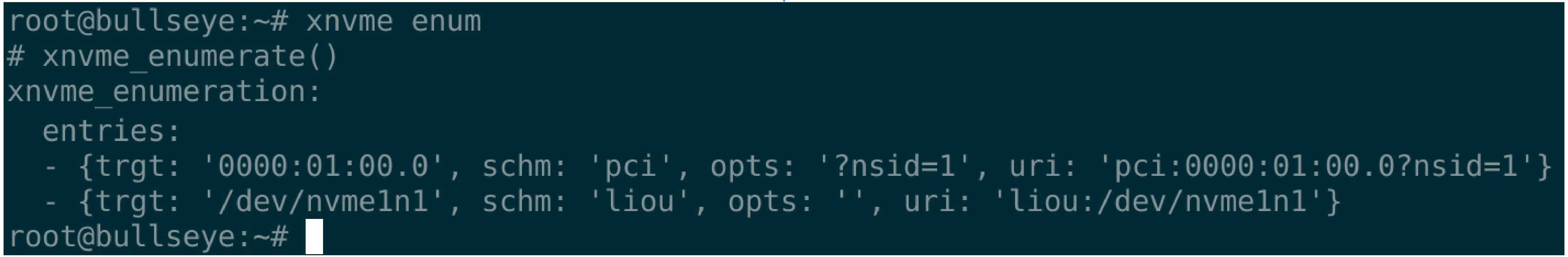
FreeBSD Kernel NVMe Driver

Programming Emerging Storage Interfaces using the xNVMe API

- **xNVMe Base API**
  - Lowest level interface
- **Device**
  - Handles
  - Identifiers
  - **Enumeration**
  - Geometry

Two devices in the system

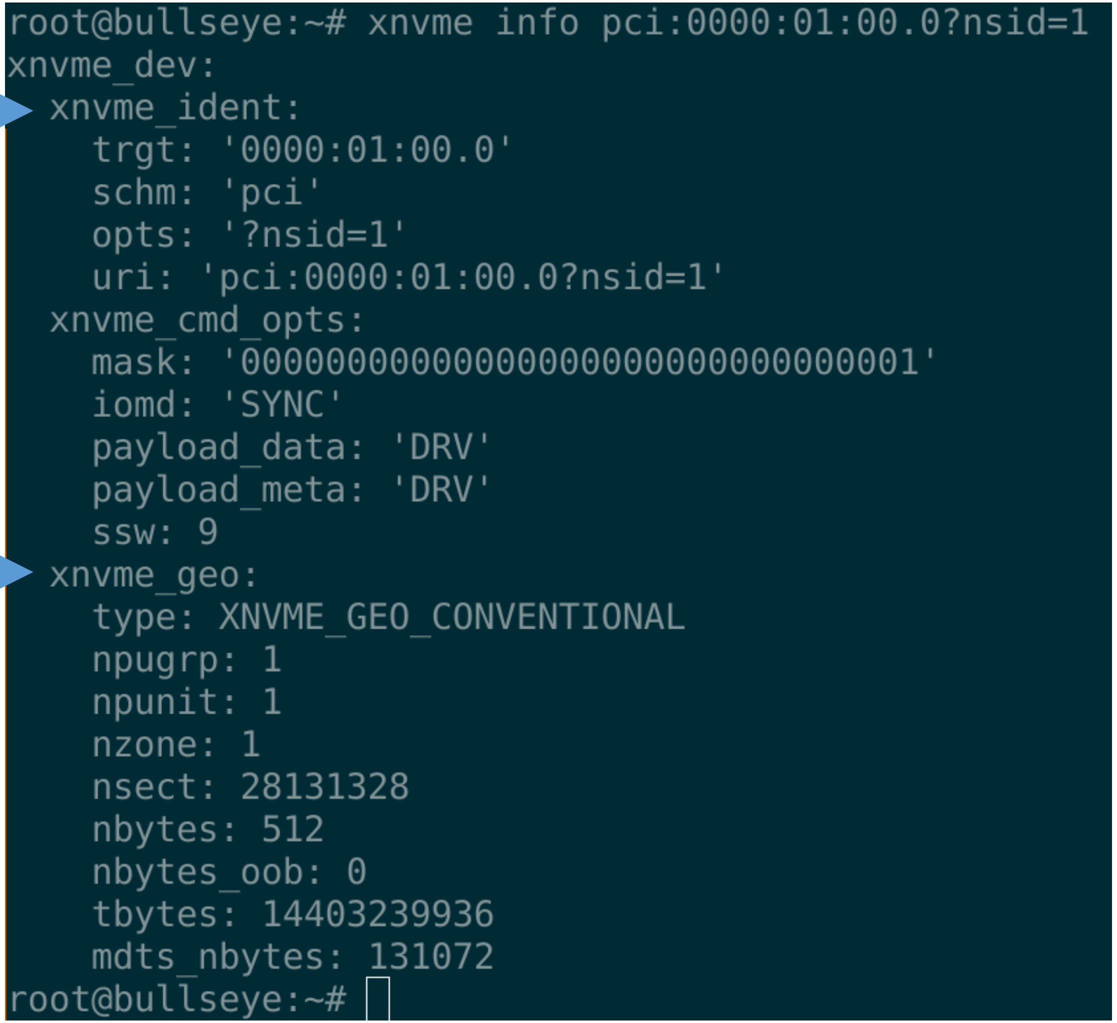
One is attached to the user space NVMe driver (**SPDK**)
The other is attached to the **Linux Kernel** NVMe Driver



```
root@bullseye:~# xnvme enum
# xnvme_enumerate()
xnvme_enumeration:
  entries:
  - {trgt: '0000:01:00.0', schm: 'pci', opts: '?nsid=1', uri: 'pci:0000:01:00.0?nsid=1'}
  - {trgt: '/dev/nvme1n1', schm: 'liou', opts: '', uri: 'liou:/dev/nvme1n1'}
root@bullseye:~#
```

Programming Emerging Storage Interfaces using the xNVMe API

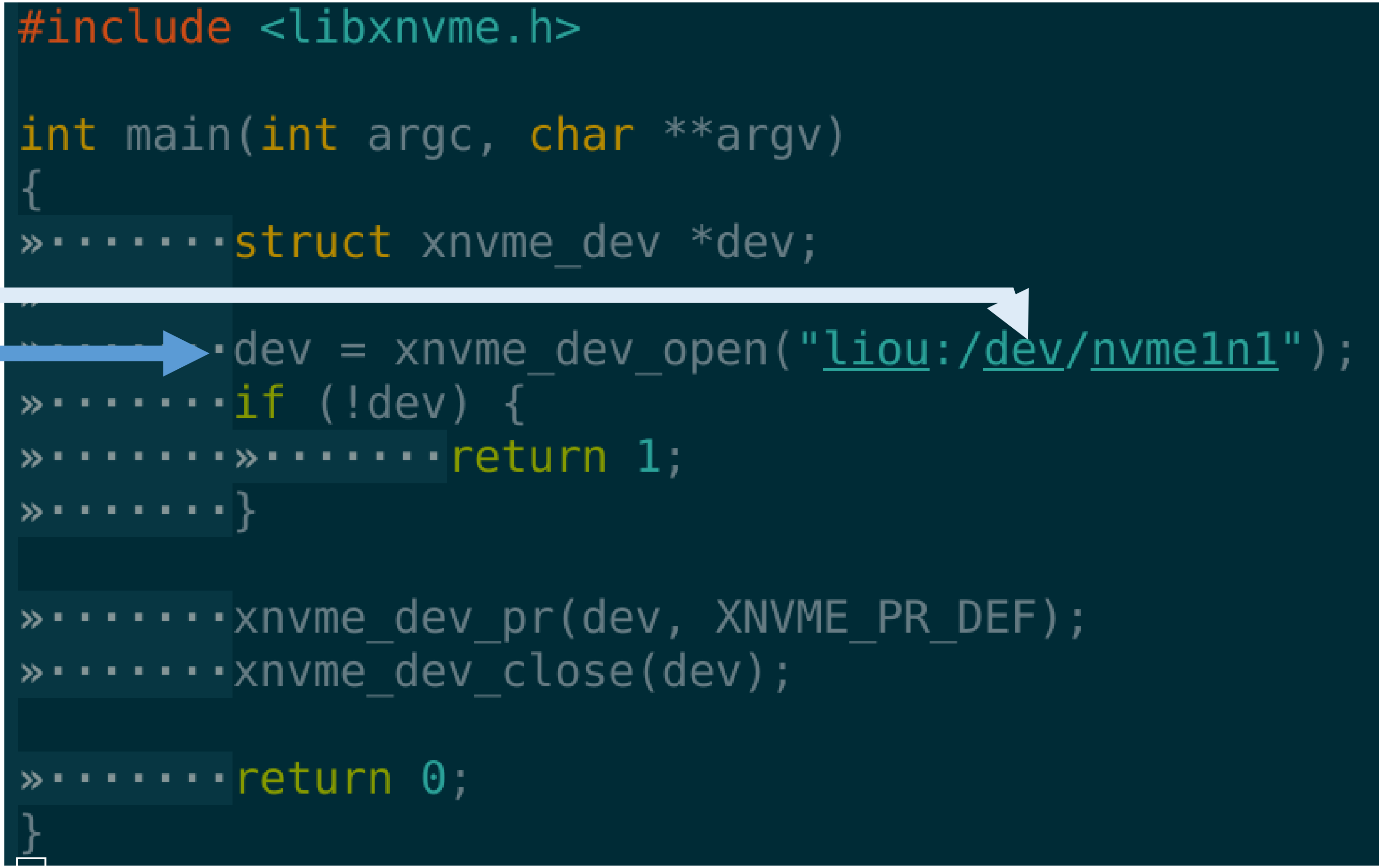
- **xNVMe Base API**
  - Lowest level interface
- **Device**
  - Handles
  - **Identifiers**
  - Enumeration
  - **Geometry**
- Memory Management
  - Command payloads
  - Virtual memory
- **Command Interface**
  - Synchronous
  - Asynchronous
    - Context and callback



```
root@bullseye:~# xnvme info pci:0000:01:00.0?nsid=1
xnvme_dev:
  xnvme_ident:
    trgt: '0000:01:00.0'
    schm: 'pci'
    opts: '?nsid=1'
    uri: 'pci:0000:01:00.0?nsid=1'
  xnvme_cmd_opts:
    mask: '00000000000000000000000000000001'
    iomd: 'SYNC'
    payload_data: 'DRV'
    payload_meta: 'DRV'
    ssw: 9
  xnvme_geo:
    type: XNVME_GEO_CONVENTIONAL
    npugrp: 1
    npunit: 1
    nzone: 1
    nsect: 28131328
    nbytes: 512
    nbytes_oob: 0
    tbytes: 14403239936
    mdts_nbytes: 131072
root@bullseye:~# 
```

Programming Emerging Storage Interfaces using the xNVMe API

- **xNVMe Base API**
  - Lowest level interface
- **Device**
  - **Handles**
  - **Identifiers**
  - Enumeration
  - Geometry
- Memory Management
  - Command payloads
  - Virtual memory
- Command Interface
  - Synchronous
  - Asynchronous
    - Context and callback



```
#include <libxnvme.h>

int main(int argc, char **argv)
{
»·······struct xnvme_dev *dev;
»
»·······dev = xnvme_dev_open("liou:/dev/nvme1n1");
»·······if (!dev) {
»·······»·······return 1;
»·······}

»·······xnvme_dev_pr(dev, XNVME_PR_DEF);
»·······xnvme_dev_close(dev);

»·······return 0;
}
```

Programming Emerging Storage Interfaces using the xNVMe API

- **xNVMe Base API**
  - Lowest level interface
- Device
  - **Handles**
  - Identifiers
  - Enumeration
  - Geometry
- **Memory Management**
  - **Command payloads**
  - **Virtual memory**
- Command Interface
  - Synchronous
  - Asynchronous
    - Context and callback

```
void *
xnvme_buf_alloc(const struct xnvme_dev *dev, size_t nbytes, uint64_t *phys);
int
xnvme_buf_vtophys(const struct xnvme_dev *dev, void *buf, uint64_t *phys);
void
xnvme_buf_free(const struct xnvme_dev *dev, void *buf);
```

When possible: the buffer-allocators will allocate physical / DMA transferable memory to achieve **zero-copy** payloads

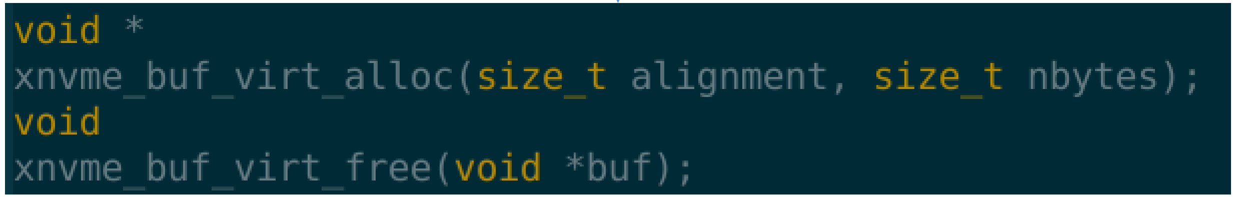
Programming Emerging Storage Interfaces using the xNVMe API

- **xNVMe Base API**
  - Lowest level interface
- Device
  - **Handles**
  - Identifiers
  - Enumeration
  - Geometry
- **Memory Management**
  - **Command payloads**
  - **Virtual memory**
- Command Interface
  - Synchronous
  - Asynchronous
    - Context and callback

```
void *
xnvme_buf_alloc(const struct xnvme_dev *dev, size_t nbytes, uint64_t *phys);
int
xnvme_buf_vtophys(const struct xnvme_dev *dev, void *buf, uint64_t *phys);
void
xnvme_buf_free(const struct xnvme_dev *dev, void *buf);
```

When possible: the buffer-allocators will allocate physical / DMA transferable memory to achieve **zero-copy** payloads

The virtual memory allocators will by default use libc but are mappable to other allocators such as TCMalloc

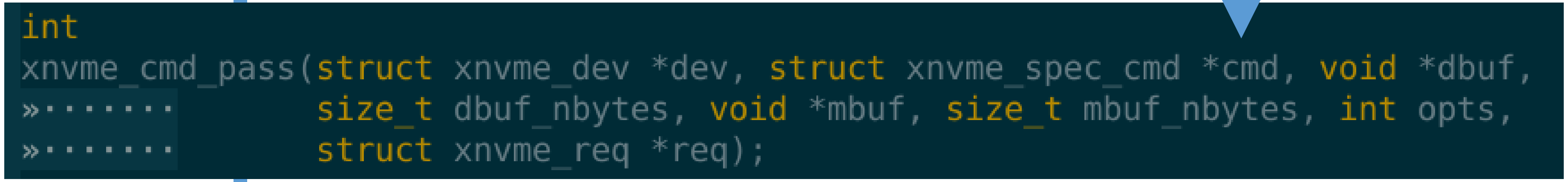


```
void *
xnvme_buf_virt_alloc(size_t alignment, size_t nbytes);
void
xnvme_buf_virt_free(void *buf);
```

Programming Emerging Storage Interfaces using the xNVMe API

- **xNVMe Base API**
  - Lowest level interface
- Device
  - **Handles**
  - Identifiers
  - Enumeration
  - Geometry
- Memory Management
  - **Command payloads**
  - Virtual memory
- **Command Interface**
  - Synchronous
  - Asynchronous
    - Context and callback

Command **Passthrough**
The user constructs the command

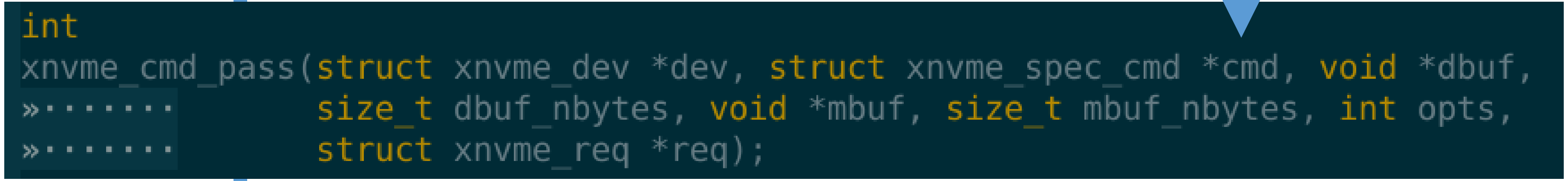


```
int
xnvme_cmd_pass(struct xnvme_dev *dev, struct xnvme_spec_cmd *cmd, void *dbuf,
»·······       size_t dbuf_nbytes, void *mbuf, size_t mbuf_nbytes, int opts,
»·······       struct xnvme_req *req);
```

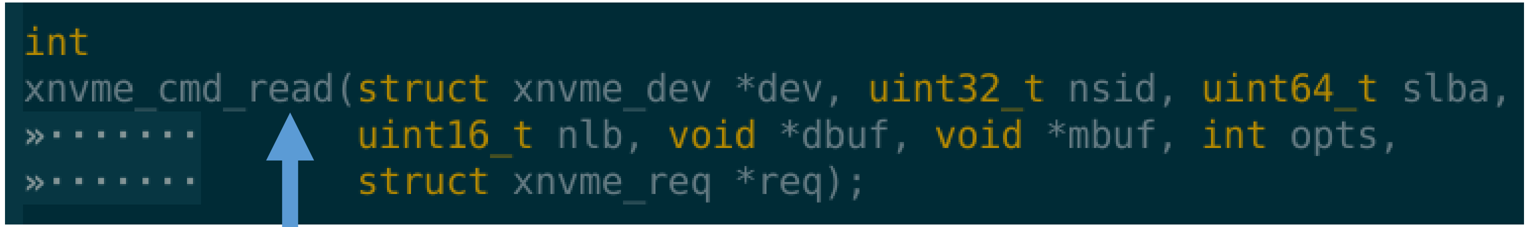
Programming Emerging Storage Interfaces using the xNVMe API

- **xNVMe Base API**
  - Lowest level interface
- Device
  - **Handles**
  - Identifiers
  - Enumeration
  - Geometry
- Memory Management
  - **Command payloads**
  - Virtual memory
- **Command Interface**
  - Synchronous
  - Asynchronous
    - Context and callback

Command **Passthrough**
The user constructs the command



```
int
xnvme_cmd_pass(struct xnvme_dev *dev, struct xnvme_spec_cmd *cmd, void *dbuf,
»·······       size_t dbuf_nbytes, void *mbuf, size_t mbuf_nbytes, int opts,
»·······       struct xnvme_req *req);
```

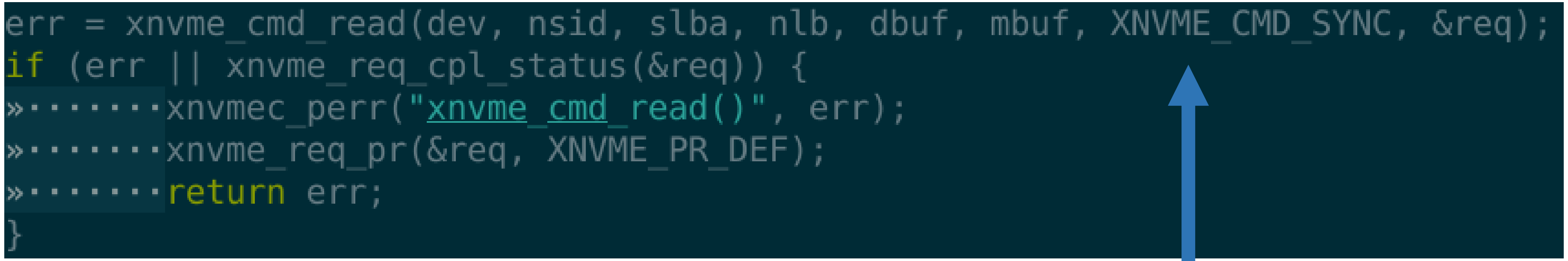


```
int
xnvme_cmd_read(struct xnvme_dev *dev, uint32_t nsid, uint64_t slba,
»·······       uint16_t nlb, void *dbuf, void *mbuf, int opts,
»·······       struct xnvme_req *req);
```

Command **Encapsulation**
The library constructs the command

Programming Emerging Storage Interfaces using the xNVMe API

- **xNVMe Base API**
  - Lowest level interface
- Device
  - **Handles**
  - Identifiers
  - Enumeration
  - Geometry
- Memory Management
  - **Command payloads**
  - Virtual memory
- **Command Interface**
  - **Synchronous**
  - Asynchronous
    - Context and callback



```
err = xnvme_cmd_read(dev, nsid, slba, nlb, dbuf, mbuf, XNVME_CMD_SYNC, &req);
if (err || xnvme_req_cpl_status(&req)) {
»·······xnvmec_perr("xnvme_cmd_read()", err);
»·······xnvme_req_pr(&req, XNVME_PR_DEF);
»·······return err;
}
```

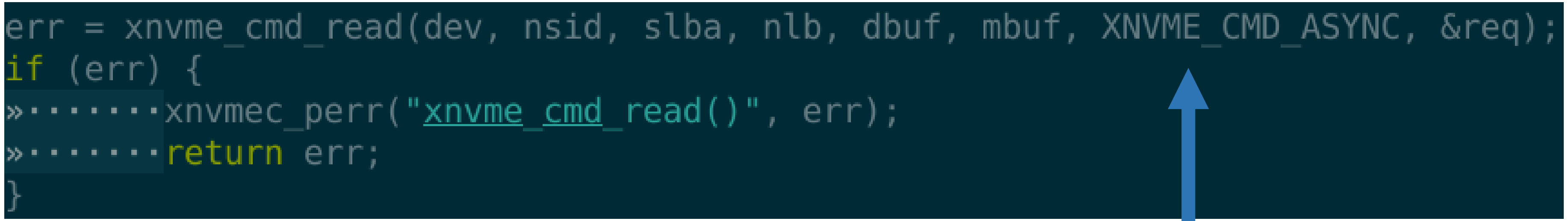
Synchronous Command Execution

Set command-option **XNVME_CMD_SYNC**
Check **err** for submission status
Check **req** for completion status

THE NEXT CREATION STARTS HERE

Programming Emerging Storage Interfaces using the xNVMe API

- **xNVMe Base API**
  - Lowest level interface
- Device
  - **Handles**
  - Identifiers
  - Enumeration
  - Geometry
- Memory Management
  - **Command payloads**
  - Virtual memory
- **Command Interface**
  - Synchronous
  - **Asynchronous**
    - Context and callback



```
err = xnvme_cmd_read(dev, nsid, slba, nlb, dbuf, mbuf, XNVME_CMD_ASYNC, &req);
if (err) {
»·······xnvmec_perr("xnvme_cmd_read()", err);
»·······return err;
}
```

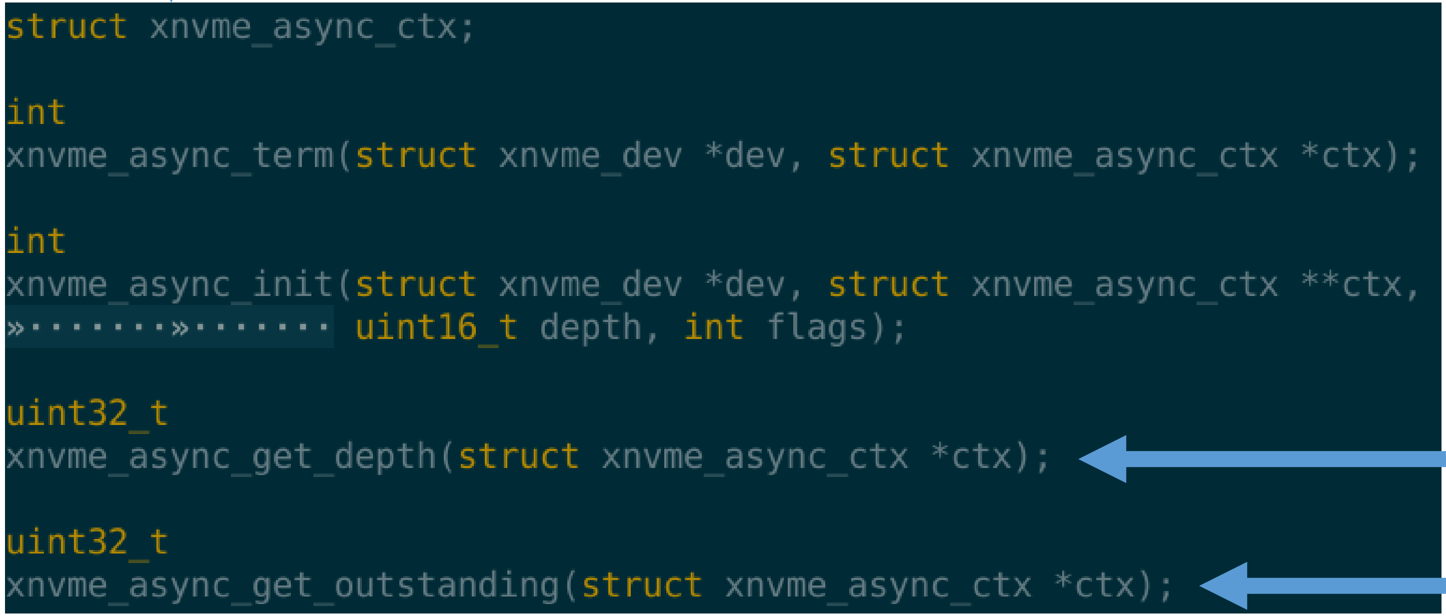
Asynchronous Command Execution

Set command-option **XNVME_CMD_ASYNC**
Check **err** for submission status
What about **completions**?

Programming Emerging Storage Interfaces using the xNVMe API

- **xNVMe Base API**
  - Lowest level interface
- Device
  - **Handles**
  - Identifiers
  - Enumeration
  - Geometry
- Memory Management
  - **Command payloads**
  - Virtual memory
- **Command Interface**
  - Synchronous
  - **Asynchronous**
    - **Context** and callback

Asynchronous Context
Opaque structure backed by an encapsulation of an **io_uring** sq/cq ring or an **SPDK** IO queue-pair.



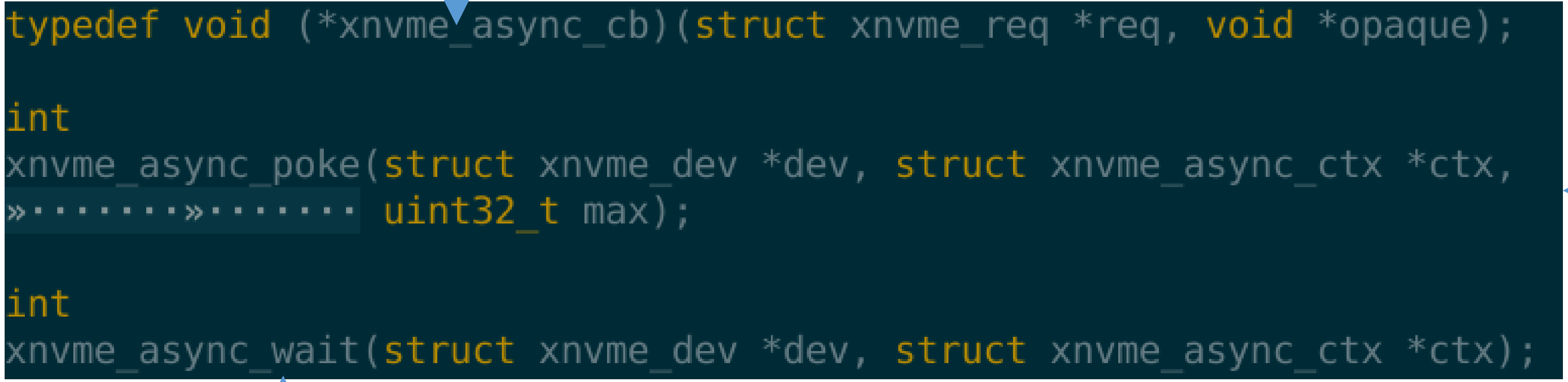
```
struct xnvme_async_ctx;

int
xnvme_async_term(struct xnvme_dev *dev, struct xnvme_async_ctx *ctx);

int
xnvme_async_init(struct xnvme_dev *dev, struct xnvme_async_ctx **ctx,
»·······»······· uint16_t depth, int flags);

uint32_t
xnvme_async_get_depth(struct xnvme_async_ctx *ctx);

uint32_t
xnvme_async_get_outstanding(struct xnvme_async_ctx *ctx);
```

Helper functions to retrieve maximum queue-depth and the current number of commands in-flight / outstanding

Programming Emerging Storage Interfaces using the xNVMe API

- **xNVMe Base API**
  - Lowest level interface
- Device
  - **Handles**
  - Identifiers
  - Enumeration
  - Geometry
- Memory Management
  - **Command payloads**
  - Virtual memory
- **Command Interface**
  - Synchronous
  - **Asynchronous**
    - **Context** and **callback**

Callback function; called upon command **completion**



```
typedef void (*xnvme_async_cb)(struct xnvme_req *req, void *opaque);

int
xnvme_async_poke(struct xnvme_dev *dev, struct xnvme_async_ctx *ctx,
»·······»······· uint32_t max);

int
xnvme_async_wait(struct xnvme_dev *dev, struct xnvme_async_ctx *ctx);
```

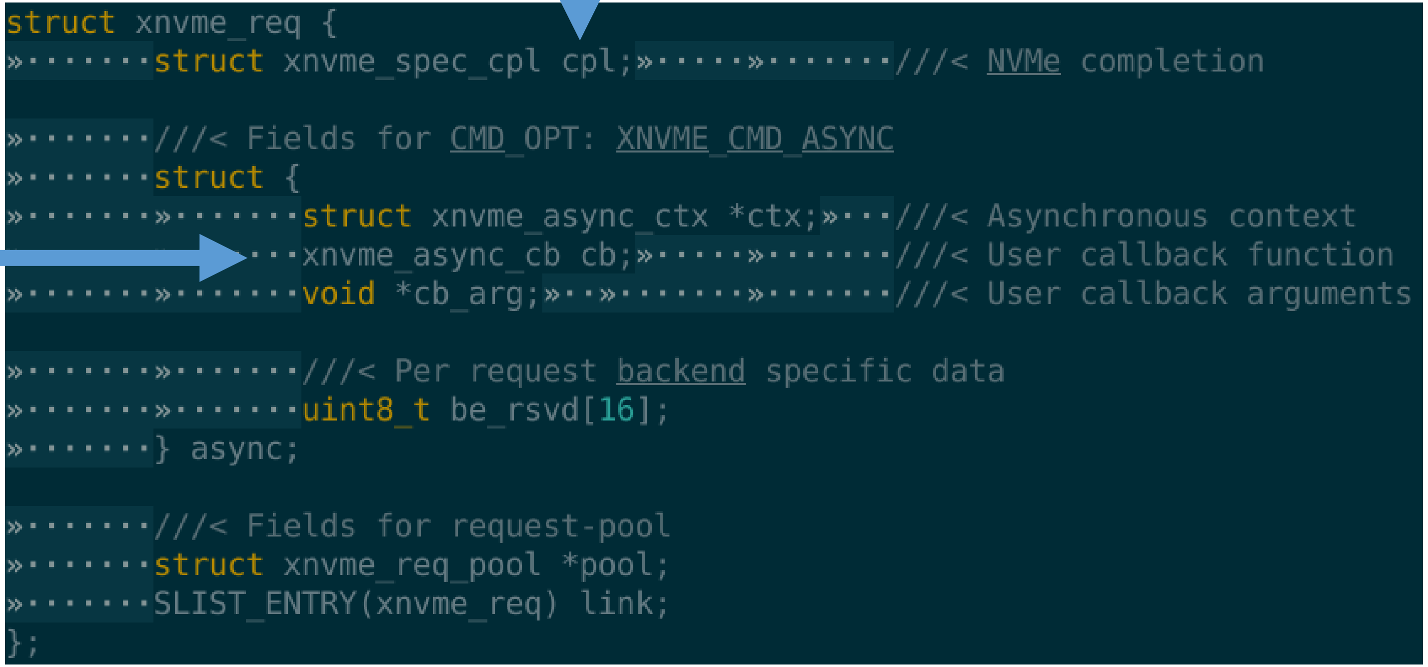
Wait, blocking, until there are no more commands outstanding on the given asynchronous context

Reap / process, at most **max**, completions, non-blocking

Programming Emerging Storage Interfaces using the xNVMe API

- **xNVMe Base API**
  - Lowest level interface
- Device
  - **Handles**
  - Identifiers
  - Enumeration
  - Geometry
- Memory Management
  - **Command payloads**
  - Virtual memory
- **Command Interface**
  - Synchronous
  - **Asynchronous**
    - **Context** and **callback**

Command completion result; used by the **synchronous** as well as the **asynchronous** command modes



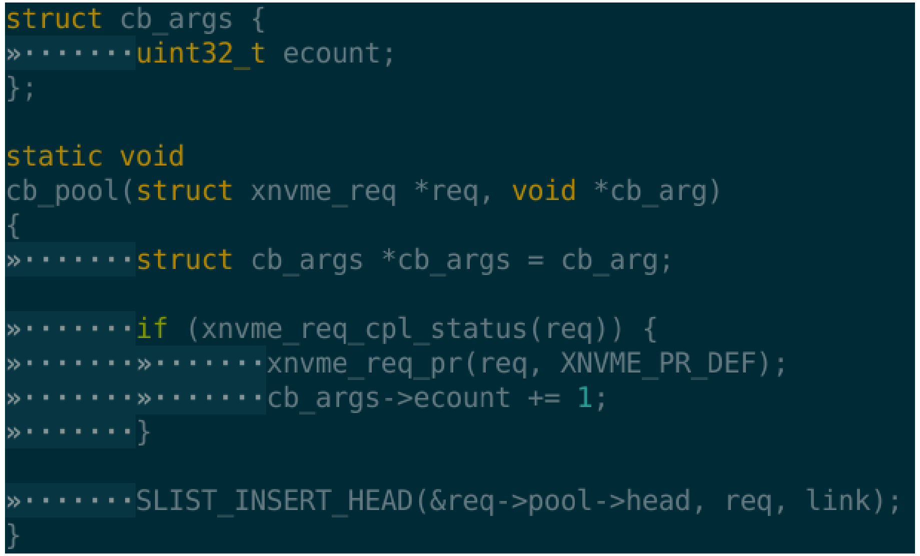
```c
struct xnvme_req {
	struct xnvme_spec_cpl cpl;		///< NVMe completion

	///< Fields for CMD_OPT: XNVME_CMD_ASYNC
	struct {
		struct xnvme_async_ctx *ctx;	///< Asynchronous context
		xnvme_async_cb cb;		///< User callback function
		void *cb_arg;			///< User callback arguments

		///< Per request backend specific data
		uint8_t be_rsvd[16];
	} async;

	///< Fields for request-pool
	struct xnvme_req_pool *pool;
	SLIST_ENTRY(xnvme_req) link;
};
```

Asynchronous fields: **context**, **callback**, and **callback-argument**

Programming Emerging Storage Interfaces using the xNVMe API

- **xNVMe Asynchronous API Example**

User-defined **callback** argument and callback **function**

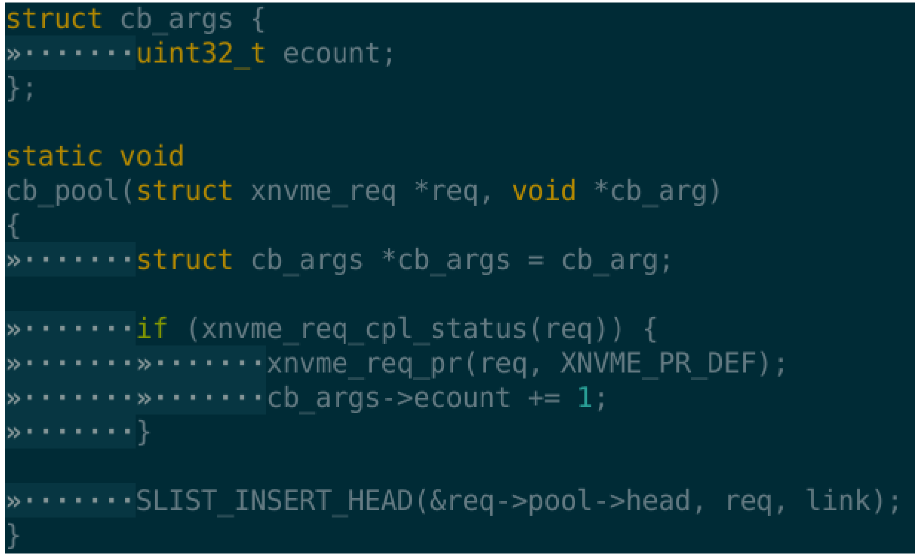


```
struct cb_args {
»·······uint32_t ecount;
};

static void
cb_pool(struct xnvme_req *req, void *cb_arg)
{
»·······struct cb_args *cb_args = cb_arg;

»·······if (xnvme_req_cpl_status(req)) {
»·······»·······xnvme_req_pr(req, XNVME_PR_DEF);
»·······»·······cb_args->ecount += 1;
»·······}

»·······SLIST_INSERT_HEAD(&req->pool->head, req, link);
}
```

Programming Emerging Storage Interfaces using the xNVMe API

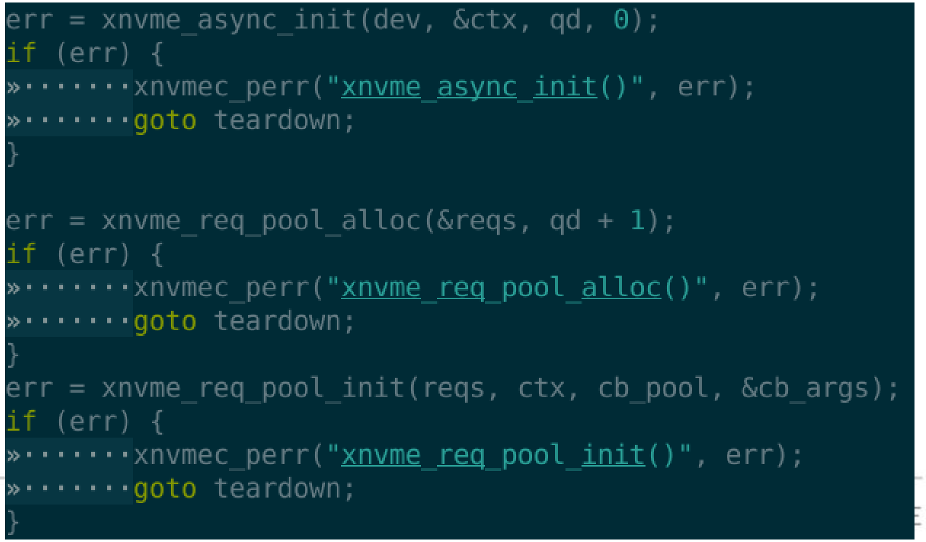
- **xNVMe Asynchronous API Example**

User-defined **callback** argument and callback **function**



```
struct cb_args {
»·······uint32_t ecount;
};

static void
cb_pool(struct xnvme_req *req, void *cb_arg)
{
»·······struct cb_args *cb_args = cb_arg;

»·······if (xnvme_req_cpl_status(req)) {
»·······»·······xnvme_req_pr(req, XNVME_PR_DEF);
»·······»·······cb_args->ecount += 1;
»·······}

»·······SLIST_INSERT_HEAD(&req->pool->head, req, link);
}
```

Asynchronous **context** and **request-pool** initialization

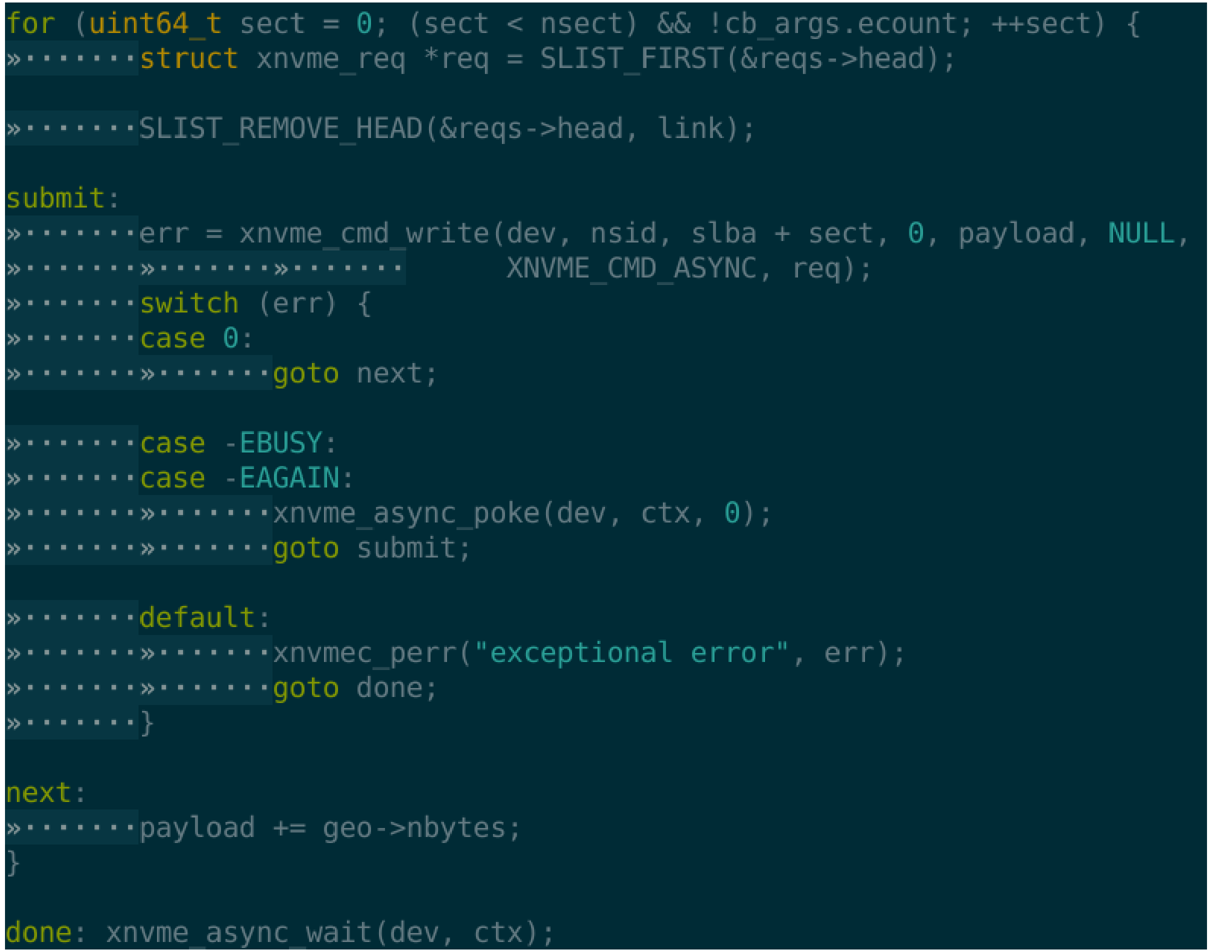


```
err = xnvme_async_init(dev, &ctx, qd, 0);
if (err) {
»·······xnvmec_perr("xnvme_async_init()", err);
»·······goto teardown;
}

err = xnvme_req_pool_alloc(&reqs, qd + 1);
if (err) {
»·······xnvmec_perr("xnvme_req_pool_alloc()", err);
»·······goto teardown;
}
err = xnvme_req_pool_init(reqs, ctx, cb_pool, &cb_args);
if (err) {
»·······xnvmec_perr("xnvme_req_pool_init()", err);
»·······goto teardown;
}
```

Programming Emerging Storage Interfaces using the xNVMe API

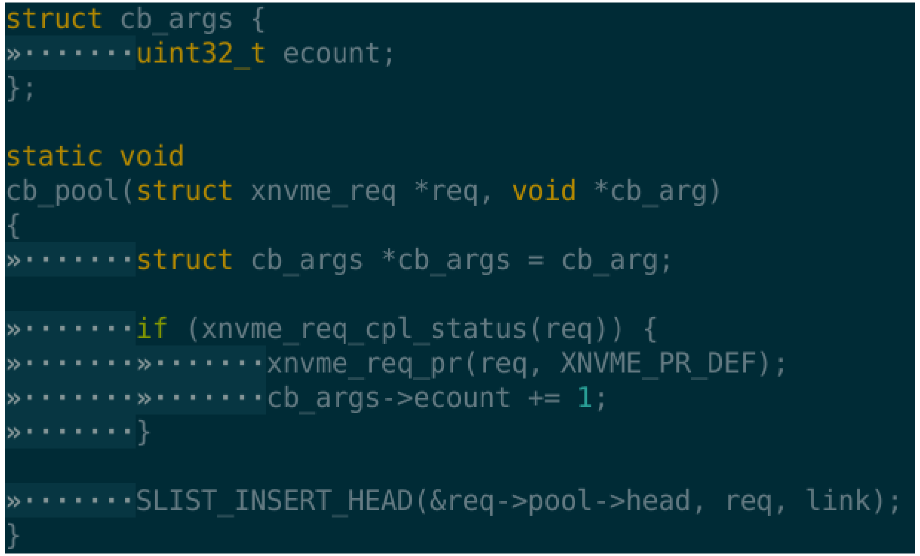
- **xNVMe Asynchronous API Example**

Writing a payload to device



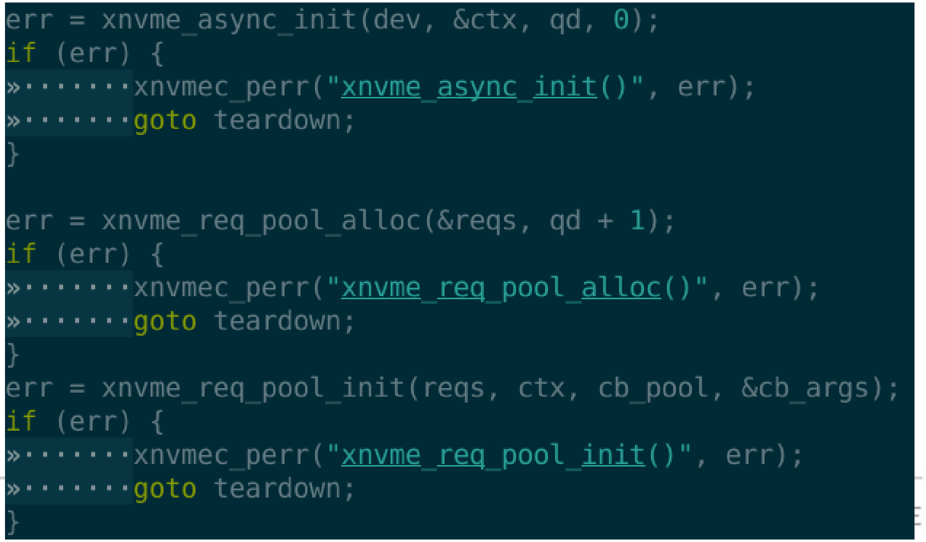
```
for (uint64_t sect = 0; (sect < nsect) && !cb_args.ecount; ++sect) {
»·······struct xnvme_req *req = SLIST_FIRST(&reqs->head);

»·······SLIST_REMOVE_HEAD(&reqs->head, link);

submit:
»·······err = xnvme_cmd_write(dev, nsid, slba + sect, 0, payload, NULL,
»·······»·······»·······          XNVME_CMD_ASYNC, req);
»·······switch (err) {
»·······case 0:
»·······»·······goto next;

»·······case -EBUSY:
»·······case -EAGAIN:
»·······»·······xnvme_async_poke(dev, ctx, 0);
»·······»·······goto submit;

»·······default:
»·······»·······xnvmec_perr("exceptional error", err);
»·······»·······goto done;
»·······}

next:
»·······payload += geo->nbytes;
}

done: xnvme_async_wait(dev, ctx);
```

User-defined **callback** argument and callback **function**



```
struct cb_args {
»·······uint32_t ecount;
};

static void
cb_pool(struct xnvme_req *req, void *cb_arg)
{
»·······struct cb_args *cb_args = cb_arg;

»·······if (xnvme_req_cpl_status(req)) {
»·······»·······xnvme_req_pr(req, XNVME_PR_DEF);
»·······»·······cb_args->ecount += 1;
»·······}

»·······SLIST_INSERT_HEAD(&req->pool->head, req, link);
}
```

Asynchronous **context** and **request-pool** initialization



```
err = xnvme_async_init(dev, &ctx, qd, 0);
if (err) {
»·······xnvmec_perr("xnvme_async_init()", err);
»·······goto teardown;
}

err = xnvme_req_pool_alloc(&reqs, qd + 1);
if (err) {
»·······xnvmec_perr("xnvme_req_pool_alloc()", err);
»·······goto teardown;
}
err = xnvme_req_pool_init(reqs, ctx, cb_pool, &cb_args);
if (err) {
»·······xnvmec_perr("xnvme_req_pool_init()", err);
»·······goto teardown;
}
```

Programming Emerging Storage Interfaces: What does it cost?

***NOTE:** System hardware, Linux Kernel, Software, NVMe Device Specs. and Null Block Device configuration in the last slide

Programming Emerging Storage Interfaces: What does it cost?

- **It is free, as in, APACHE 2.0**

***NOTE:** System hardware, Linux Kernel, Software, NVMe Device Specs. and Null Block Device configuration in the last slide

Programming Emerging Storage Interfaces: What does it cost?

- **It is free, as in, APACHE 2.0**
- Evaluating potential **efficiency*** cost of using **xNVMe**
  - Cost in terms of nanoseconds per command **aka** layer-overhead
  - Benchmark using fio **4K** Random Read at **QD1**
  - Compare the regular (**REGLR**) interface to **xNVMe**
- Using a **physical** locally attached NVMe device
- Using a Linux Null Block instance **without** the block-layer

***NOTE:** System hardware, Linux Kernel, Software, NVMe Device Specs. and Null Block Device configuration in the last slide

Programming Emerging Storage Interfaces: What does it cost?

- **It is free, as in, APACHE 2.0**
- Evaluating potential **efficiency*** cost of using **xNVMe**
  - Cost in terms of nanoseconds per command **aka** layer-overhead
  - Benchmark using fio **4K** Random Read at **QD1**
  - Compare the regular (**REGLR**) interface to **xNVMe**
- Using a **physical** locally attached NVMe device

Comparing 🍎 to 🍎	Latency (**nsec**)
REGLR/io_uring +SQT +RF	**8336**
xNVMe/io_uring +SQT +RF	**8373**
Overhead	**~36**

***NOTE:** System hardware, Linux Kernel, Software, NVMe Device Specs. and Null Block Device configuration in the last slide

Programming Emerging Storage Interfaces: What does it cost?

- **It is free, as in, APACHE 2.0**
- Evaluating potential **efficiency*** cost of using **xNVMe**
  - Cost in terms of nanoseconds per command **aka** layer-overhead
  - Benchmark using fio **4K** Random Read at **QD1**
  - Compare the regular (**REGLR**) interface to **xNVMe**
- Using a **physical** locally attached NVMe device

Comparing 🍎 to 🍎	Latency (**nsec**)
REGLR/io_uring +SQT +RF	**8336**
xNVMe/io_uring +SQT +RF	**8373**
Overhead	**~36**

Comparing 🍊 to 🍊	Latency (**nsec**)
REGLR/SPDK	**6471**
xNVMe/SPDK	**6510**
Overhead	**~39**

***NOTE:** System hardware, Linux Kernel, Software, NVMe Device Specs. and Null Block Device configuration in the last slide

Programming Emerging Storage Interfaces: What does it cost?

- **It is free, as in, APACHE 2.0**
- Evaluating potential **efficiency*** cost of using **xNVMe**
  - Cost in terms of nanoseconds per command **aka** layer-overhead
  - Benchmark using fio **4K** Random Read at **QD1**
  - Compare the regular (**REGLR**) interface to **xNVMe**
- Using a **physical** locally attached NVMe device

Comparing 🍎 to 🍎	Latency (**nsec**)
REGLR/io_uring +SQT +RF	**8336**
xNVMe/io_uring +SQT +RF	**8373**
Overhead	**~36**

Comparing 🍊 to 🍊	Latency (**nsec**)
REGLR/SPDK	**6471**
xNVMe/SPDK	**6510**
Overhead	**~39**

➔ Overhead about 36-39 **nsec**

***NOTE:** System hardware, Linux Kernel, Software, NVMe Device Specs. and Null Block Device configuration in the last slide

Programming Emerging Storage Interfaces: What does it cost?

- **It is free, as in, APACHE 2.0**
- Evaluating potential **efficiency*** cost of using **xNVMe**
  - Cost in terms of nanoseconds per command **aka** layer-overhead
  - Benchmark using fio **4K** Random Read at **QD1**
  - Compare the regular (**REGLR**) interface to **xNVMe**
- Using a **physical** locally attached NVMe device ➔ 36-39 **nsec**
- Using a Linux Null Block instance **without** the block-layer

***NOTE:** System hardware, Linux Kernel, Software, NVMe Device Specs. and Null Block Device configuration in the last slide

Programming Emerging Storage Interfaces: What does it cost?

- **It is free, as in, APACHE 2.0**
- Evaluating potential **efficiency*** cost of using **xNVMe**
  - Cost in terms of nanoseconds per command **aka** layer-overhead
  - Benchmark using fio **4K** Random Read at **QD1**
  - Compare the regular (**REGLR**) interface to **xNVMe**
- Using a **physical** locally attached NVMe device ➔ 36-39 **nsec**
- Using a Linux Null Block instance **without** the block-layer

Comparing 🥕 to 🥕	Latency (**nsec**)
REGLR/io_uring +SQT +RF	**644**
xNVMe/io_uring +SQT +RF	**730**
Overhead	**86**

➔ Overhead about 86 **nsec**

***NOTE:** System hardware, Linux Kernel, Software, NVMe Device Specs. and Null Block Device configuration in the last slide

Programming Emerging Storage Interfaces: What does it cost?

- **It is free, as in, APACHE 2.0**
- Evaluating potential **efficiency*** cost of using **xNVMe**
  - Cost in terms of nanoseconds per command **aka** layer-overhead
  - Benchmark using fio **4K** Random Read at **QD1**
  - Compare the regular (**REGLR**) interface to **xNVMe**
- Using a **physical** locally attached NVMe device ➔ 36-39 **nsec**
- Using a Linux Null Block instance **without** the block-layer ➔ 86 **nsec**
- Where is time spent?
  - Function wrapping and pointer indirection
  - Popping + pushing requests from pool
  - Callback invocation
  - Pseudo io_vec is filled and consumes space (io_uring)
  - Suboptimal request-struct layout

***NOTE:** System hardware, Linux Kernel, Software, NVMe Device Specs. and Null Block Device configuration in the last slide

Programming Emerging Storage Interfaces: What does it cost?

- **It is free, as in, APACHE 2.0**
- Evaluating potential **efficiency*** cost of using **xNVMe**
  - Cost in terms of nanoseconds per command **aka** layer-overhead
  - Benchmark using fio **4K** Random Read at **QD1**
  - Compare the regular (**REGLR**) interface to **xNVMe**
- Using a **physical** locally attached NVMe device ➔ 36-39 **nsec**
- Using a Linux Null Block instance **without** the block-layer ➔ 86 **nsec**
- Where is time spent?
  - Function wrapping and pointer indirection
  - Popping + pushing requests from pool
  - Callback invocation
  - Pseudo io_vec is filled and consumes space (io_uring)
  - Suboptimal request-struct layout

Things an application it likely to require when doing more than synthetically re-submitting upon completion

***NOTE:** System hardware, Linux Kernel, Software, NVMe Device Specs. and Null Block Device configuration in the last slide

Programming Emerging Storage Interfaces: What does it cost?

- **It is free, as in, APACHE 2.0**
- Evaluating potential **efficiency*** cost of using **xNVMe**
  - Cost in terms of nanoseconds per command **aka** layer-overhead
  - Benchmark using fio **4K** Random Read at **QD1**
  - Compare the regular (**REGLR**) interface to **xNVMe**
- Using a **physical** locally attached NVMe device ➔ 36-39 **nsec**
- Using a Linux Null Block instance **without** the block-layer ➔ 86 **nsec**
- Where is time spent?
  - Function wrapping and pointer indirection
  - Popping + pushing requests from pool
  - Callback invocation
  - Pseudo io_vec is filled and consumes space (io_uring)
  - Suboptimal request-struct layout

Things an application it likely to require when doing more than synthetically re-submitting upon completion

Things that need fixing

***NOTE:** System hardware, Linux Kernel, Software, NVMe Device Specs. and Null Block Device configuration in the last slide

Programming Emerging Storage Interfaces: What does it cost?

- **It is free, as in, APACHE 2.0**
- **Current cost, about 40~90 nanoseconds per command**
  - About the same cost as a DRAM load
  - Cost less than **not** enabling **`IORING_REGISTER_BUFFERS`** (~100nsec)
  - Cost less than going through a PCIe switch (~150nsec)
  - Cost a fraction of going through the block layer (~1850nsec)
  - Cost a lot less than a read from todays **fast** media (~8000nsec)

Programming Emerging Storage Interfaces: What does it cost?

- **It is free, as in, APACHE 2.0**
- **Current cost, about 40~90 nanoseconds per command**
  - About the same cost as a DRAM load
  - Cost less than **not** enabling **`IORING_REGISTER_BUFFERS`** (~100nsec)
  - Cost less than going through a PCIe switch (~150nsec)
  - Cost a fraction of going through the block layer (~1850nsec)
  - Cost a lot less than a read from todays **fast** media (~8000nsec)

➔ **Cost will go down!**

Programming Emerging Storage Interfaces: What does it cost?

- **It is free, as in, APACHE 2.0**
- **Current cost, about 40~90 nanoseconds per command**
  - About the same cost as a DRAM load
  - Cost less than **not** enabling **`IORING_REGISTER_BUFFERS`** (~100nsec)
  - Cost less than going through a PCIe switch (~150nsec)
  - Cost a fraction of going through the block layer (~1850nsec)
  - Cost a lot less than a read from todays **fast** media (~8000nsec)

  ➔ **Cost will go down!**

Re-target your application without changes	IOPS	MB/s
`./`**`your_app`**` pci:0000:01.00?nsid=1`	**150 K**	**613**
`./`**`your_app`**` /dev/nvme0n1`	**116 K**	**456**

- **What do you get?**
  - An even *easier* API
    - High-level abstractions when you need them
    - Peel of the layers and get low-level **control** when you do not
  - Your applications, tools, and libraries will run on Linux, FreeBSD, and SPDK

Programming Emerging Storage Interfaces: What does it cost?

- **It is free, as in, APACHE 2.0**
- **Current cost, about 40~90 nanoseconds per command**

  ➔ Cost will go down!

Re-target your application without changes	IOPS	MB/s
`./your_app pci:0000:01.00?nsid=1`	**150 K**	**613**
`./your_app /dev/nvme0n1`	**116 K**	**456**

- **What do you get?**
  - An even *easier* API
    - High-level abstractions when you need them
    - Peel of the layers and get low-level **control** when you do not
  - Your applications, tools, and libraries will run on Linux, FreeBSD, and SPDK
- **There is more!**

Programming Emerging Storage Interfaces: What does it cost?

- **It is free, as in, APACHE 2.0**
- **Current cost, about 40~90 nanoseconds per command**

  ➔ Cost will go down!

Re-target your application without changes	IOPS	MB/s
`./your_app pci:0000:01.00?nsid=1`	**150 K**	**613**
`./your_app /dev/nvme0n1`	**116 K**	**456**

- **What do you get?**
  - An even *easier* API
    - High-level abstractions when you need them
    - Peel of the layers and get low-level **control** when you do not
  - Your applications, tools, and libraries will run on Linux, FreeBSD, and SPDK
- **There is more!**
  - On top of the Base API: Command-set APIs e.g. Zoned Namespaces
  - NVMe Meta File System – browse logs as files in binary and YAML
  - Command-line tool builders (library and bash-completion generator)

Programming Emerging Storage Interfaces: What does it cost?

- **It is free, as in, APACHE 2.0**
- **Current cost, about 40~90 nanoseconds per command**

  ➔ Cost will go down!

Re-target your application without changes	IOPS	MB/s
`./your_app pci:0000:01.00?nsid=1`	**150 K**	**613**
`./your_app /dev/nvme0n1`	**116 K**	**456**

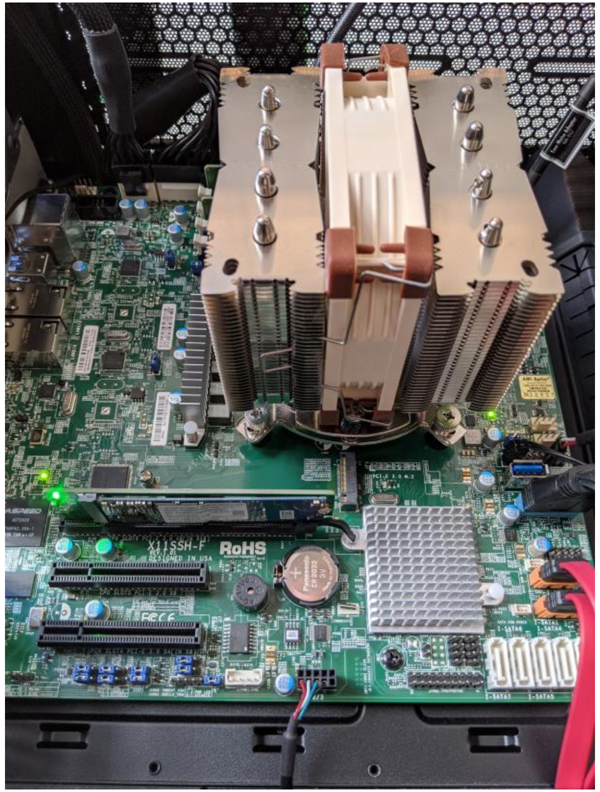
- **What do you get?**
  - An even *easier* API
    - High-level abstractions when you need them
    - Peel of the layers and get low-level **control** when you do not
  - Your applications, tools, and libraries will run on Linux, FreeBSD, and SPDK
- **There is more!**
  - On top of the Base API: Command-set APIs e.g. Zoned Namespaces
  - NVMe Meta File System – browse logs as files in binary and YAML
  - Command-line tool builders (library and bash-completion generator)
- First release: https://xnvme.io Q1 2020

THE NEXT CREATION STARTS HERE

Placing **memory** at the forefront of future innovation and creative IT life

Programming Emerging Storage Interfaces: test rig

- Slides, logs and numbers will be made available on: https://xnvme.io



- **System Spec**
  - Supermicro X11SSH-F
  - Intel Xeon E3-1240 v6 @ 3.7Ghz
  - 2x 16GB DDR4 2667 Mhz
- **Software**
  - **Debian Linux 5.4.13-1 / fio 3.17 / liburing Feb. 14. 2020**
  - **xNVMe 0.0.14 / SPDK v19.10.x / fio 3.3 (SPDK plugin)**
- **NVMe Device Specs.**

	Latency	IOPS	BW
Random Read	8 **usec**	190 K	900 MB/sec
Random Write	30 **usec**	35 K	150 MB/sec

- **Null Block Device Config (bio-based)**

```
queue_mode=0 irqmode=0 nr_devices=1 completion_nsec=10 home_node=0 gb=100 bs=512 submit_queues=1
hw_queue_depth=64 use_per_node_hctx=0 no_sched=0 blocking=0 shared_tags=0 zoned=0 zone_size=256 zone_nr_conv=0
```

- **Null Block Device Config (mq)**

```
queue_mode=1 irqmode=0 nr_devices=1 completion_nsec=10 home_node=0 gb=100 bs=512 submit_queues=1
hw_queue_depth=64 use_per_node_hctx=0 no_sched=0 blocking=0 shared_tags=0 zoned=0 zone_size=256 zone_nr_conv=0
```